TU WIEN Informatics

Foundations of Bitcoin-Compatible Scalability Protocols

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Lukas Aumayr, BSc
Registration Number 01325536

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Matteo Maffei
Second advisor: Prof. Pedro Moreno-Sanchez, IMDEA Software Institute, Spain

The dissertation has been reviewed by:

Prof. Arthur Gervais	Prof. Roger Wattenhofer

Vienna, February 12, 2024

Lukas Aumayr

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Lukas Aumayr, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Februar 2024

Lukas Aumayr

To my parents, Gertrud and Friedrich.

Acknowledgements

I would like to express my deep gratitude to Matteo Maffei, whose guidance throughout my PhD has been nothing short of transformative. His relentless optimism, passion for research, and commitment to advancing my academic and career prospects have always lifted me up, especially during the more difficult parts of this journey. His unwavering belief in my abilities and his consistent efforts to highlight my work have greatly enhanced my academic progress.

Equally, I owe a great debt of gratitude to Pedro Moreno-Sanchez. When I began my PhD journey, despite my non-cryptographic background, he offered me the invaluable gift of the benefit of the doubt. His belief in my potential, his dedication to my best interests, and his invaluable insights were instrumental in giving my PhD the direction and momentum it needed. Our academic dialogues, morning runs, cherished breakfasts, and dinners were highlights of my PhD and created lasting memories.

I also want to thank CoBloX for generously funding my PhD including my numerous travels to conferences during my PhD, and especially Philipp Hoenisch for our engaging discussions. His insights, especially from a more pragmatic perspective, were enlightening, and our conversations about novel challenges were always a source of inspiration.

My collaboration and interactions with colleagues have been a cornerstone of this journey. I extend my heartfelt thanks to Oğuzhan Ersoy, Siavash Riahi, Kristina Hostáková, Andreas Erwig, Sebastian Faust, Kasra Abbaszadeh, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Iosif Salem, Stefan Schmid, Giulia Scaffino, Zeta Avarikioti, Subhra Mazumdar, Mahsa Bastankhah, and Dionysis Zindros. Each of them has left a mark on my work and thoughts, enriching this journey in countless ways.

A special nod to the Security and Privacy research group at TU Wien. Our combined efforts, from rigorous discussions to light-hearted gatherings, have fostered a wonderful academic community. I fondly remember the activities from climbing, game nights, jam sessions, and countless other events that made my time in the group so memorable.

During my academic voyage, I had opportunities to spend time at the IMDEA Software Institute in Madrid, the DFINITY Foundation in Zurich, and Stanford University. At IMDEA, the camaraderie and shared leisurely pursuits with Pedro, Dimitrios Vasilopoulos, and the rest of the group added a wonderful dimension to my stay and made it particularly memorable. At DFINITY, I am incredibly grateful to Gregory Neven and David Derler

for their enthusiasm and mentorship during my internship. My time at Stanford was made especially memorable thanks to the great company of Giulia, Dionysis, Joachim Neu, and Srivatsan Sridhar.

In addition, I cherish the bonds and insightful interactions I have formed with fellow researchers during numerous conferences. These connections, though unnamed here, have played a pivotal role in shaping my academic perspective and opening doors for future collaborations.

Outside the confines of academia, my circle of friends has been a beacon of support and joy. To Markus, Reinhold, Tom, David, Peter, Nici and Adrian: your consistent support, combined with countless shared moments, made the stressful moments of this journey bearable.

Finally, I must emphasize the fundamental role played by my family. My parents, the unwavering pillars of support, have been the bedrock upon which this achievement stands. The love and encouragement of my brother and sister, their spouses, my grandparents, and my nieces have been a guiding light throughout this journey. Throughout life's ups and downs, my family has kept everything in perspective.

Kurzfassung

Permissionless Blockchains ermöglichen es Nutzern, die sich gegenseitig misstrauen, Geld auf dezentrale Weise zu senden. Leider stoßen diese Blockchains auf ein Skalierbarkeitsproblem, was bedeutet, dass sie technisch nur eine vergleichsweise geringe Anzahl von Transaktionen im Vergleich zu traditionellen, zentralisierten Systemen verarbeiten können. *Payment Channel Networks* (PCNs) gehören zu den bekanntesten Lösungen, um diese Skalierbarkeitsprobleme zu lösen.

Die Grundidee von PCNs besteht darin, Transaktionen in sogenannte *Zahlungskanäle* zwischen zwei Nutzern auszulagern und diese Kanäle dann zu vernetzen, sodass jedes Paar von Benutzern, welches durch einen Pfad von Kanälen verbunden ist, Transaktionen durchführen kann. Der Vorteil besteht darin, dass nur die Transaktionen zum Öffnen und Schließen dieser Kanäle in der Blockchain gespeichert werden müssen, während alle anderen Transaktionen außerhalb der Blockchain stattfinden, wodurch der Transaktionsdurchsatz insgesamt erhöht wird. Es gibt verschiedene PCN-Protokolle, die in der Praxis eingesetzt werden (z.B. das Lightning Network mit einem Wert von ca. 150 Mio. USD). Aber auch diese haben eine Reihe von Problemen.

In dieser Arbeit untersuchen wir bestehende PCN-Protokolle und identifizieren Probleme in Bezug auf *Sicherheit*, *Datenschutz*, *Effizienz* und *eingeschränkte Funktionalität*. Gleichzeitig stellen wir neue Protokolle vor, die diese Probleme überwinden und den aktuellen Stand der Technik verbessern. Wir konzentrieren uns auf Bitcoin-kompatible Lösungen, da Bitcoin nicht nur die Kryptowährung mit der größten Marktkapitalisierung ist, sondern auch über ein begrenztes Skripting-Set verfügt, wodurch unsere Protokolle mit vielen anderen Kryptowährungen kompatibel sind. Wir führen auch eine strenge formale Sicherheitsanalyse unserer Protokolle durch. Konkret leistet diese Arbeit die folgenden Beiträge.

Zuerst führen wir *Sleepy Channels* ein, die sichere Zahlungskanäle ermöglichen, auch wenn Benutzer nicht durchgehend online sind. Dies ist ein bedeutender Unterschied zu bestehenden Konstruktionen, bei denen die Gelder der Nutzer gefährdet sind, wenn sie offline sind. Darüber hinaus verallgemeinern wir den Begriff der Zahlungskanäle (*Generalized Channels*) und sorgen dafür, dass sie jede Anwendung unterstützen, die von der zugrunde liegenden Blockchain unterstützt wird, und nicht nur Zahlungen.

Zweitens führen wir eine neue Konstruktion (*Blitz*) ein, die sichere Zahlungen über einen Pfad mit mehreren Kanälen in PCNs ermöglicht und dabei die Anzahl der Interaktionen

für jeden Vermittler auf eine einzige reduziert (von zwei oder mehr) und nur konstante Zeit benötigt, anstatt linear in der Pfadlänge. Wir bieten auch die erste sichere Konstruktion für die atomare Aktualisierung mehrerer Kanäle, die nicht auf einem Pfad liegen (*Thora*).

Schließlich stellen wir die erste Bitcoin-kompatible Konstruktion für virtuelle Kanäle bereit (*Bitcoin-Compatible Virtual Channels*). Mit diesen virtuellen Kanälen können zwei Benutzer über einen Vermittler einen direkten Kanal öffnen, ohne eine Öffnungs- oder Schließtransaktion in die Blockchain aufzunehmen. Darüber hinaus analysieren wir andere Konstruktionen für virtuelle Kanäle, identifizieren einen neuartigen Angriff und führen eine sichere und effiziente Konstruktion ein, die über mehrere Vermittler funktioniert (*Donner*).

Diese Beiträge greifen nahtlos ineinander. Gemeinsam bieten sie eine vielseitige Ad-hoc-Lösung, die zwei beliebige Benutzer sicher miteinander verbindet, ohne einen Fußabdruck auf der Blockchain zu hinterlassen, für Anwendungen, die über Zahlungen hinausgehen. Diese Arbeit zielt darauf ab, das Verständnis von PCNs neu zu gestalten und allgemeinere und effizientere Lösungen für das Skalierbarkeitsproblem zu bieten.

Abstract

Permissionless blockchains allow mutually untrusted users to transfer money in a decentralized way. Unfortunately, these blockchains face a scalability problem, which means they are technically limited to processing only a relatively small number of transactions compared to traditional, centralized systems. *Payment Channel Networks* (PCNs) are among the most prominent solutions to mitigate these scalability issues.

The basic idea of PCNs is to outsource transactions to so-called *payment channels* between two users and then link these channels to form a network where any two users connected by a path of channels can perform transactions. The advantage is that only the transactions for opening and closing these channels need to go on the blockchain, while any other transaction happens outside of the blockchain, thus increasing the overall transaction throughput. Several different PCN protocols exist and are used in practice (e.g., the Lightning Network with a value of approximately 150M USD). However, even these have their sets of issues.

In this thesis, we investigate existing PCN protocols and identify issues in terms of *security*, *privacy*, *efficiency*, and *limited functionality*. Simultaneously, we introduce new protocols that overcome these issues and improve the state of the art. We focus on Bitcoin-compatible solutions since Bitcoin is not only the largest cryptocurrency in terms of market capitalization but also has a limited set of scripting capabilities, thus making our protocols compatible with a large number of other cryptocurrencies as well. We also conduct a rigorous formal security analysis of our protocols. More concretely, this thesis makes the following contributions.

First, we introduce *Sleepy Channels*, enabling secure payment channels even when users are not continuously online. This is a significant shift from existing constructions where going offline puts users' funds at risk. Further, we generalize the notion of payment channels (*Generalized Channels*) and make them support any application that the underlying blockchain supports rather than only payments.

Second, we introduce a new construction (*Blitz*) that achieves secure payments across a path of multiple channels in PCNs while reducing the number of interactions for each intermediary to a single one (from two or more) and takes only constant time instead of linear in the path length. We also provide the first secure construction for atomically updating multiple channels that are not on a path (*Thora*).

Finally, we provide the first *Bitcoin-Compatible Virtual Channel* construction. These virtual channels allow two users to open a direct channel via one intermediary without putting an opening or closing transaction on the blockchain. We further analyze other existing virtual channel constructions, identify a novel attack, and introduce secure and efficient virtual channels over multiple intermediaries (*Donner*).

These contributions interplay seamlessly. Collectively, they offer a versatile, ad-hoc solution connecting any two users securely without an on-chain footprint for applications that go beyond payments. Thus, this thesis aims to reshape the understanding of PCNs and give more general and efficient solutions to the scalability problem.

List of Publications

Papers in Conferences with Proceedings.

[ATM+22] Lukas Aumayr, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 179–192. Association for Computing Machinery, 2022. **Part of this thesis.**

[AEE+21] Lukas Aumayr, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures. In *Advances in Cryptology – ASIACRYPT 2021*, pages 635–664. Springer International Publishing, 2021. **Part of this thesis.**

[AMSKM21] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4043–4060. USENIX Association, 2021. **Part of this thesis.**

[AAM22] Lukas Aumayr, Kasra Abbaszadeh, and Matteo Maffei. Thora: Atomic and Privacy-Preserving Multi-Channel Updates. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 165–178. Association for Computing Machinery, 2022. **Part of this thesis.**

[AME+21] Lukas Aumayr, Matteo Maffei, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. Bitcoin-Compatible Virtual Channels. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 901–918. IEEE Computer Society, 2021. **Part of this thesis.**

[AMSKM23] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Breaking and Fixing Virtual Channels: Domino Attack and Donner. In

30th Annual Network and Distributed System Security Symposium, NDSS 2023. The Internet Society, 2023. **Part of this thesis.**

[SAAM22] Giulia Scaffino, Lukas Aumayr, Zeta Avarikioti, and Matteo Maffei. Glimpse: On-demand Light Client with Constant-size Storage for DeFi. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 733–750. USENIX Association, 2023.

Contents

Kurzfassung **ix**

Abstract **xi**

List of Publications **xiii**

Contents **xv**

1 Introduction **1**
1.1 Overview of Payment Channel Networks 3
1.2 State of the Art and Limitations 7
1.3 Methodology 12
1.4 Contributions 13

2 Sleepy Channels: Bi-directional Payment Channels without Watchtowers **17**
2.1 Introduction 18
2.2 Solution Overview 24
2.3 Preliminaries 28
2.4 Ideal Functionality Bi-directional Channels 29
2.5 Sleepy Channels: Our Bi-Directional Payment Channel Protocol . . . 32
2.6 Performance Evaluation 38
2.7 Conclusion 41

3 Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures **43**
3.1 Introduction 44
3.2 Background and Solution Overview 49
3.3 Preliminaries 53
3.4 Generalized channels 54
3.5 Adaptor Signatures 59
3.6 Generalized Channel Construction 64
3.7 Applications 68
3.8 Performance Analysis 71

4 Blitz: Secure Multi-Hop Payments Without Two-Phase Commits 73
4.1 Introduction . . . 74
4.2 Background and notation . . . 77
4.3 Solution overview . . . 81
4.4 Our construction . . . 88
4.5 Security analysis . . . 93
4.6 Evaluation . . . 95
4.7 Related work . . . 98
4.8 Conclusion . . . 98

5 Thora: Atomic and Privacy-Preserving Multi-Channel Updates 101
5.1 Introduction . . . 102
5.2 Background . . . 105
5.3 Solution overview . . . 109
5.4 Construction . . . 114
5.5 Security analysis . . . 121
5.6 Evaluation . . . 124
5.7 Applications . . . 126
5.8 Discussion . . . 128
5.9 Conclusion . . . 129

6 Bitcoin-Compatible Virtual Channels 131
6.1 Introduction . . . 132
6.2 Background . . . 135
6.3 Virtual Channels . . . 140
6.4 Security Model and Analysis . . . 151
6.5 Performance evaluation . . . 152
6.6 Related Work . . . 156
6.7 Conclusion . . . 157

7 Breaking and Fixing Virtual Channels: Domino Attack and Donner 159
7.1 Introduction . . . 160
7.2 Background and notation . . . 165
7.3 The Domino attack . . . 170
7.4 Donner: Key ideas . . . 175
7.5 Donner: Protocol description . . . 177
7.6 Security analysis . . . 183
7.7 Evaluation and comparison . . . 184
7.8 Conclusion . . . 187

8 Conclusion and Directions of Future Research 189
8.1 Conclusion . . . 189
8.2 Directions for Future Work . . . 190

List of Figures **193**

List of Tables **201**

Bibliography **205**

A Appendix to Chapter 2 **221**
A.1 UC Protocol . . . 221
A.2 Deployment cost . . . 236

B Appendix to Chapter 3 **237**
B.1 On the Usage of the UC-Framework . . . 237
B.2 Schnorr-based Adaptor Signature . . . 240
B.3 Proof of the ECDSA-based Adaptor Signature . . . 255
B.4 Pre-signature unforgeability . . . 268
B.5 Additional material to generalized channel protocol . . . 269
B.6 Simplifying functionality description . . . 274
B.7 Simplifying the protocol descriptions . . . 276
B.8 Security proof . . . 277
B.9 Applications on top of generalized channels . . . 283

C Appendix to Chapter 4 **287**
C.1 Discussion on practical deployment . . . 287
C.2 1-phase commits in distributed databases . . . 289
C.3 Payment channels in more detail . . . 290
C.4 Preventing the race condition when the sender is irrational . . . 291
C.5 Concrete attack scenarios (informal) . . . 291
C.6 Timeline . . . 292
C.7 Communication overhead . . . 292
C.8 Extended simulation results . . . 293
C.9 Extended macros . . . 293
C.10 Modeling in the UC framework . . . 297
C.11 Discussion on security and privacy goals . . . 316

D Appendix to Chapter 5 **321**
D.1 Stealth addresses . . . 321
D.2 UC modeling . . . 322
D.3 Discussion on security and privacy . . . 346

E Appendix to Chapter 6 **349**
E.1 On the usage of the UC-Framework . . . 349
E.2 Adaptor Signatures . . . 352
E.3 Additional material to ledger channels . . . 353
E.4 Virtual Channels . . . 362
E.5 Wrappers for Missing Checks . . . 382

F Appendix to Chapter 7 **403**
F.1 When to use virtual channels 403
F.2 Extended comparison and discussion 406
F.3 Operation examples 407
F.4 Extended background 407
F.5 Extended macros, prerequisites and protocol 410
F.6 UC modeling 414

CHAPTER 1

Introduction

The seminal 2009 Bitcoin whitepaper [Nak09] gave rise to permissionless cryptocurrencies. This innovative form of digital cash forgoes the need for trusted, centralized authorities such as (central) banks, governments, or other intermediaries. Instead, anyone with a computer connected to the Internet can participate and conduct payments in a trustless way. This is achieved through a decentralized protocol, where participants store valid transactions in a global, distributed, immutable ledger, typically a blockchain, and run a consensus mechanism, where they agree on the validity and ordering of all transactions.

Existing permissionless cryptocurrencies, i.e., cryptocurrencies where everyone can become a full protocol participant, face a scalability problem. For instance, Bitcoin is technically limited to processing only around ten transactions per second, and it takes around one hour for transactions to be considered finalized [CDE+16]. This is a significant gap to more centralized payment systems, e.g., the VISA credit network can handle transaction loads on the order of tens of thousands of transactions per second [Tri13], finalized almost instantly.

Unfortunately, these issues are a consequence of the permissionless setting and cannot be rectified merely by increasing the number of transactions in a block or by decreasing the block generation time. Such an approach would lead to an increase in the size of the blockchain. On the one hand, this makes it harder and harder for users to become full protocol participants and thus increase centralization. On the other hand, an increased block size leads to increased latency, which, possibly together with a decreased block generation time, makes it more likely to create diverging views of the ledger, also known as *forks* [GKL15].

The scalability problem has attracted attention from academia and industry, and different proposals exist on how to tackle this problem. New consensus mechanisms aim to increase the transaction throughput of the blockchain directly, but drawbacks include more centralization, a permissioned consensus, decreased security, additional assumptions,

or not being backward compatible with existing blockchains, leading to a fork of the network (e.g., [PKF+18, KRDO17, DFKP15, NTT22, BHK+20, MJS+14, PS17]). Another idea, *sharding*, aims to enhance scalability by partitioning the network into smaller groups (e.g., [ZMR18]). The goal of *sidechains* is to outsource transactions to another blockchain, but the sidechain requires its own consensus mechanism (e.g., [BCD+14]).

In contrast to these approaches are *off-chain* solutions, in which the bulk of transactions are handled outside of the blockchain on what is referred to as layer-2. This layer-2 is built on top of the blockchain, which is then referred to as layer-1. *Payment channel networks* (PCNs) [PD16] allow users to conduct off-chain payments within a network of 2-party channels. In *commit-chains* or *plasma* [GMSR+20, KZF+18, PB17], an operator periodically collects and submits a commitment to off-chain transactions to the blockchain. In *rollups* (e.g., [Rol, ZK-]), transaction data is posted on-chain in addition to the commitments, which leads to improved data availability, but increased on-chain overhead. There are optimistic rollups, where there is a time window to dispute incorrect transactions, and ZK-rollups, where the correctness of the included transactions is proven with SNARKs.

The advantage of off-chain solutions is that they do not require modifying the underlying consensus. They are thus compatible with existing cryptocurrencies and can be deployed without changing existing cryptocurrencies, and the underlying consensus mechanism does not limit their throughput because transactions are not posted on-chain. To achieve scalability for existing blockchains, we focus on off-chain solutions in this thesis rather than sharding, sidechains, or new consensus protocols.

Optimistic rollups (and plasma) require a long wait time (or *challenge period*) before users can withdraw funds, usually 7-14 days, which allows users to dispute state commitments before they are considered finalized [Smi23]. Furthermore, the improved data availability of optimistic rollups again leads to an on-chain storage overhead that is linear in the number of transactions. This limits the effectiveness of rollups as a scaling solution to a factor of around 100x. For example, in Ethereum, rollups would enable around 4.8k transactions per second at most compared to 45 transactions per second without rollups [But21]. ZK-rollups do not require this waiting period, as SNARKs are used to prove the validity of state commitments, which can be expensive to compute. Due to these drawbacks and the fact that rollups are not compatible with Bitcoin, we focus in particular on PCNs in this thesis.

Relevance of Bitcoin-Compatible Solutions. Bitcoin and many other cryptocurrencies have limited scripting capabilities. That means they provide only a very limited set of programming instructions (also known as `op_codes`), e.g., verifying one or multiple digital signatures or enforcing a timelock, and can host non-Turing-complete smart contracts. This contrasts blockchains such as Ethereum, which can host (quasi-)Turing-complete smart contracts.

We acknowledge that a significant research effort is dedicated to designing applications built on top of these more expressive smart contracts. Such efforts either make use

of the native capabilities of blockchains such as Ethereum, or else try to bring these capabilities to Bitcoin. Examples of the latter include the aforementioned side-chains, constructions such as BitVM [Lin23], etc. Nonetheless, designing protocols that do not rely on Turing-complete scripting in the underlying blockchain is of theoretical and practical interest as well.

As of June 2023, Bitcoin alone makes up more than 48% of the total market capitalization of cryptocurrencies, and many other cryptocurrencies are forks of Bitcoin or very similar (e.g., Bitcoin Cash, BitcoinSV, Litecoin, Dogecoin, etc.) having the same functionality or even more limited scripting capabilities (e.g., Monero, Ripple, etc.). This means that it is highly relevant in practice to reduce the scripting requirements of protocols, making them compatible with a larger number of existing and potential future cryptocurrencies. However, identifying the minimal scripting requirements to achieve certain protocols is also of theoretical interest.

Furthermore, having protocols rely on smart contracts with Turing-complete instructions and more complex ways of interacting with them (e.g., events, fallback functions, etc.) makes them harder to analyze and potentially more error-prone. These errors can be exploited, causing immense economic damage, as exemplified in the (in)famous DAO hack in Ethereum [Sie16].

Consensus-Agnostic Solutions. The environmental impact of Bitcoin cannot be overlooked; its significant energy consumption and corresponding carbon footprint pose an important challenge. The primary cause of this is Bitcoin's underlying consensus mechanism, *Proof-Of-Work*, where users expend computational power to generate blocks. Although estimates indicate that a considerable portion of Bitcoin mining utilizes renewable energy sources [Cou23], the research efforts to reduce the overall power consumption remain crucial. This has led to alternative consensus mechanisms that could potentially mitigate environmental impact, such as *Proof-Of-Stake* (e.g., [KRDO17]) and *Proof-Of-Space* [DFKP15, PKF+18].

We emphasize that even though the constructions presented in this thesis are compatible with Bitcoin, they are agnostic to the underlying consensus. Our findings can thus be deployed to a wide range of blockchains, including those that prioritize lower energy consumption. Finally, we note that in Bitcoin, the processing of transactions is technically independent of the Proof-Of-Work mechanism. Thus by boosting the transaction throughput, our contributions could theoretically help to make Bitcoin more energy-efficient, at least in terms of energy per transaction assuming the same total energy consumption.

1.1 Overview of Payment Channel Networks

Payment Channels. A payment channel is a two-party protocol that allows two users, Alice and Bob, to perform off-chain transactions. Initially, Alice and Bob can *open* a payment channel by locking some funds in a joint account on-chain, e.g., an address that requires both their signatures to be spent (*multisig* address), in what is

called a *funding transaction*. Subsequently, Alice and Bob can *update* the channel by creating and exchanging transactions off-chain that spend from the funding transaction and redistribute the funds in some new way (the new *state* of the channel). Finally, they can *close* the channel when they no longer need it by posting the latest one of these off-chain update transactions on the blockchain. We illustrate this in Figure 1.1.

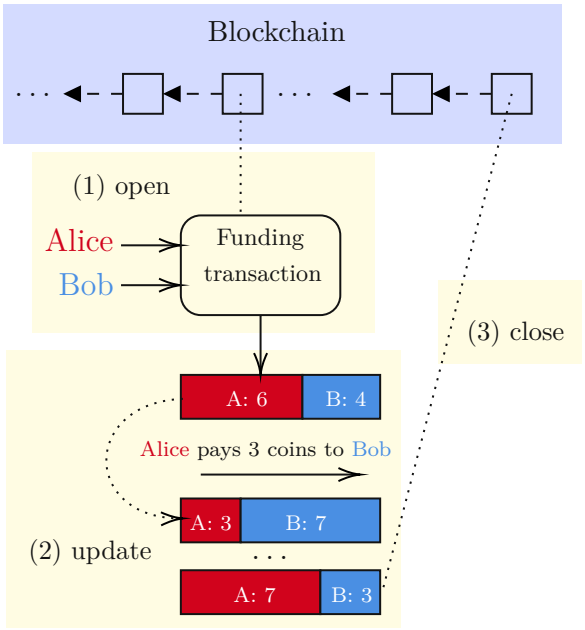

Figure 1.1: Payment channel: A payment channel consists of three operations. Alice and Bob can (1) open a payment channel by creating a *funding transaction*, which locks some of their money in a shared account (multisig address). In this example, Alice puts 6 coins and Bob 4 coins into the funding transaction. They create another transaction (state), which pays them back their coins in this initial balance distribution $(A:6, B:4)$ and then post the funding transaction on the blockchain. Then, Alice and Bob can (2) update their channel from $(A:6, B:4)$ to, e.g., $(A:3, B:7)$, which represents Alice paying 3 coins to Bob. They can continue updating their channel as often as they want. When they are done, they finally (3) close their channel with the latest state, unlocking their coins. Only two transactions go on-chain.

With this protocol, Alice and Bob can conduct arbitrarily many transactions, while only two transactions end up on the blockchain, one for opening and one for closing the channel. While this improves transaction throughput between two users, creating a payment channel for every (potential) counterparty is infeasible because every payment channel requires some money to be locked up in the funding transaction. This, in turn, leads to a large number of on-chain transactions and fees, as well as opportunity costs for the locked-up money that cannot be used otherwise or users simply not having enough funds to even open enough channels. This is why a more clever way of connecting users has emerged, i.e., Payment Channel Networks (PCNs).

Payment Channel Networks. The idea of PCNs is to establish a network of channels. However, instead of connecting all pairs of users (forming a clique), a more sparse network is formed, where typical users can have only a few channels or even a single one. The PCN can be thought of as a graph where the nodes are users, and the edges are payment channels. Typically, this graph is still connected, i.e., any two users can reach one another through a path of channels. Two nodes can transact by routing payments through these paths. This is known as multi-hop payment (MHP). The challenge here is to *synchronize* these channels on the path, such that a payment is atomically carried out and no honest user is at risk of losing their funds. This idea is illustrated in Figure 1.2. One concrete way of implementing this in practice is shown in more detail later in Section 1.2.2 and Figure 1.6.

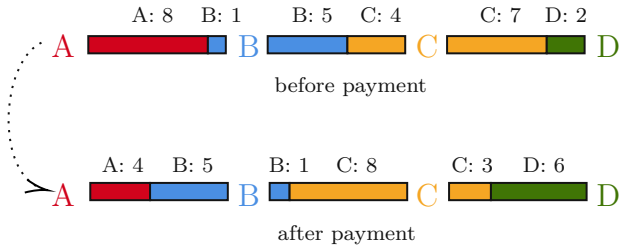

Figure 1.2: Multi-hop payment (MHP): Sender A pays 4 coins to receiver D via B and C. The colored boxes connecting two users represent payment channels (as shown in Figure 1.1). The first row shows the initial balance distribution of the channels. The second row shows the desired outcome after the payment, where 4 coins were transferred to the receiver. The challenge of MHPs is to update these channels atomically in order to prevent honest users from losing their coins.

PCNs are a feasible scaling solution and are used in practice. Most notably, the Lightning Network (LN) has around 16k nodes, 74k channels, and a total capacity of 5.4k BTC (roughly 152M USD) as of April 2023. Still, there are some fundamental drawbacks to using MHPs that essentially stem from the fact that individual payments are routed via intermediaries [AMSKM23]: (i) the intermediary users need to be online; (ii) payments are less reliable since intermediary users must actively partake in each individual payment; (iii) intermediaries charge fees per payment; (iv) each hop increases the latency of the payment, which can be up to one day per channel in the LN; (v) every intermediary learns the value that is being transacted; and (vi) they can process only a limited number of concurrent payments (e.g., 483 in the Lightning Network). Note that these drawbacks are independent of the concrete MHP implementation, but come from the fact that in MHPs, intermediaries route each individual payment. Moreover, MHPs are a synchronization protocol for payments. One would have to design a new synchronization protocol to realize applications in a multi-hop setting other than payments, such as Discreet Log Contracts (DLCs) [Dry17]. This is both an inconvenience and not trivial, as seen in a recent discussion about implementing DLCs over multiple hops [DLC21].

Virtual Channels. To overcome the drawbacks of MHPs, bypassing the intermediaries

for individual payments is crucial. To this end, Virtual Channels (VCs) can directly bridge the gap between two endpoints connected by a path of PCs. A VC is akin to a PC, but it is opened off-chain on top of the existing PCN topology instead of being opened on-chain. In other words, this approach provides the advantages of the direct connection that PCs provide with respect to scalability, but without the associated on-chain fees for opening and closing them.

To understand how VCs work, say that initially, two users, Alice and Carol, are connected by one intermediary user, Bob. Alice and Carol need to coordinate with Bob, locking some collateral in both PCs (Alice-Bob) and (Bob-Carol) for the VC. This collateral is used to compensate honest users in case one or even both of the counterparties deviate from the honest protocol execution. Otherwise, honest users get their own collateral back. This VC idea can be extended to multiple intermediary users. Any intermediary needs to be involved only in the opening and, later, the closing of the channel. However, the two endpoints can use the VC without the involvement of intermediary users. We illustrate this idea in Figure 1.3.

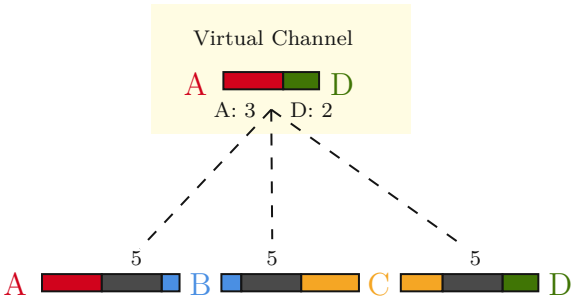

Figure 1.3: Virtual Channel (VC): A VC is not funded on-chain, but built on top of payment channels. For this, some funds of these underlying payment channels are locked (illustrated in gray) as collateral, such that the whole VC capacity is covered (in this example, 5 coins). This collateral is used to compensate honest users in case of misbehavior. After successfully closing a VC, its latest balance will be reflected in the underlying payment channels. All operations happen off-chain and while the VC is open, the two endpoints, A and D, can transact without the involvement of the intermediaries B and C.

Utilizing such a VC, the two endpoints can resolve the previously mentioned issues (i)-(vi). Intuitively, this is because intermediaries are no longer part of every payment but only part of opening and closing the VC. The endpoints can, on their own, conduct as many off-chain transactions directly, unencumbered by the increased latency, fees, decreased reliability, or privacy that goes along with MHPs. Furthermore, since VCs can be used in the same way as PCs, applications that can be built on PCs, such as DLCs, can also be built on VCs.

1.2 State of the Art and Limitations

1.2.1 State of the Art and Limitations in Payment Channels

Once payment channels are opened, users want to exchange transactions off-chain, which is also called updating the channel to a new state. Unfortunately, once a state has been signed and is valid, these signatures cannot be invalidated. The main challenge in designing payment channels is to prevent users from posting an older state, where they would receive more money than in the current state. There exist many different protocols with different trade-offs.

CLTV [Tod] and Spillman channels [Spi] propose unidirectional payment channels making use of a *replace by incentive* mechanism to deal with old states. I.e., in a channel between Alice and Bob, payments can only be made from Alice to Bob. Consequently, Bob always prefers the most recent channel state and is economically disincentivized from posting an older state. These channels have a fixed expiry set at the time of creation to prevent funds from staying locked forever. Another mechanism to handle old states is known as *replace by timelock*. This allows for bidirectional channels. Each state is time-locked and can only be posted after a pre-defined time in the future. Subsequent states have a timelock that is shorter by some safety time gap, such that the latest state is the one that can be posted first in the future. An obvious drawback is the limited number of states due to the ever-decreasing timelock.

Duplex channels [DW15] combine replace by incentive and replace by timelock and allow for bidirectional channels but have limitations on the number of payments and require an increased number of transactions to close the channel. Lightning channels [PD16] are used widely in practice. They are bidirectional and support an unlimited number of payments. The posting of old states is prevented by exchanging revocation secrets (*replace by revocation*), which in turn allows honest parties to punish misbehaving parties. This mechanism requires relative timelocks and users to be constantly online (as explained later) and is illustrated in Figure 1.4. Other bidirectional payment channel proposals, such as Eltoo [DRO], support bidirectional payments but have specific requirements for compatibility with existing cryptocurrencies, e.g., by relying on new op_codes that are as of yet not supported by Bitcoin. State channels [DFH18, DEFM19, CCF+21], which support arbitrary conditional payments, rely on complex scripts like smart contracts and are incompatible with UTXO-based currencies. Eltoo and state channels ensure the latest state via *replace by version*: Each state is marked with a sequence number, and only the state with the highest number is accepted.

Limitation L1 (Online Assumption). One fundamental issue with bidirectional payment channels that allow an unbounded number of payments, such as Lightning, Eltoo, or state channels, is that they rely on an online assumption. In other words, users must stay online and actively monitor the blockchain in order to avoid losing their funds. More concretely, if Alice is offline, Bob can post an old state that gives him more money than he should have, and Alice will miss her window of punishing Bob (Lightning) or

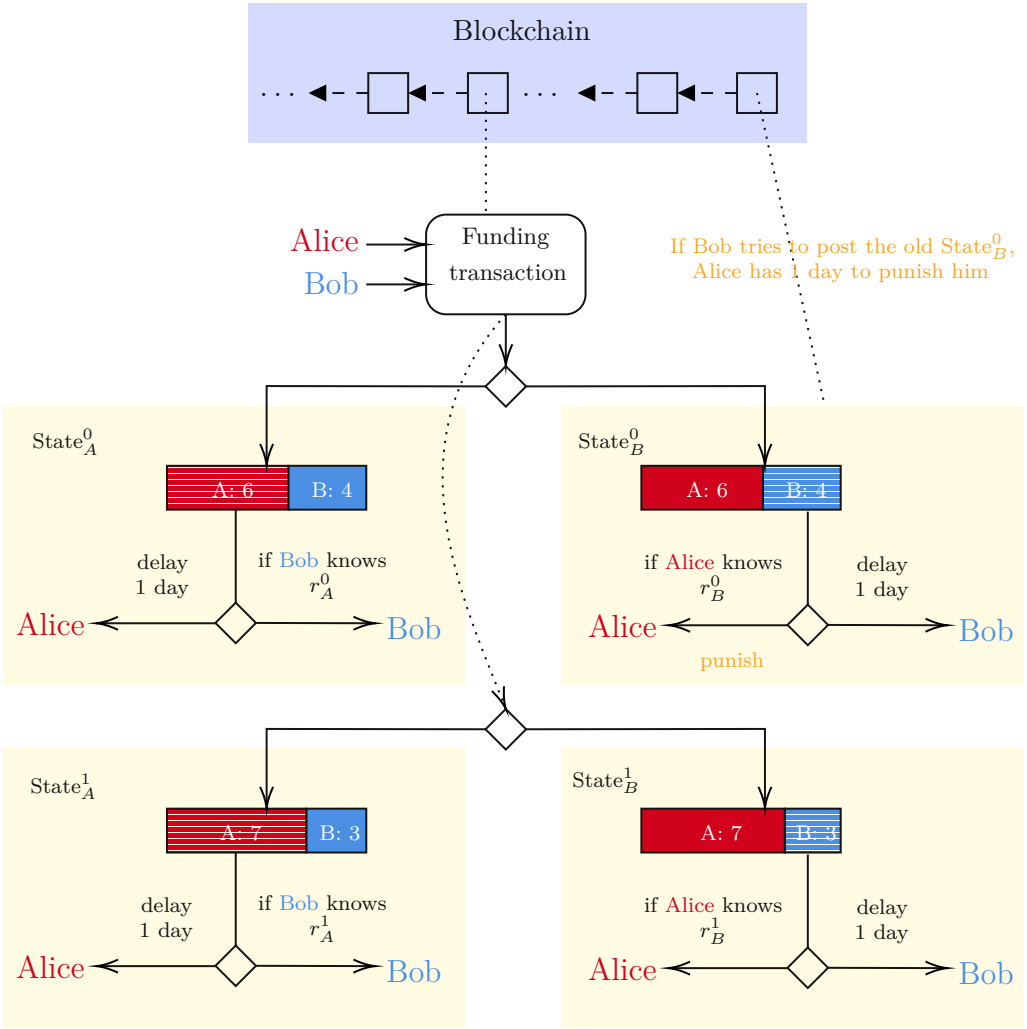

Figure 1.4: Lightning channel: In Lightning channels, there exist two versions of each state (*state duplication*), one for Alice (e.g., State^0_A) and one for Bob (e.g., State^0_B). When updating to a new state, Alice and Bob sample new revocation secrets r^1_A and r^1_B uniformly at random, respectively. Then, they create the two versions of the new state State^1_A and State^1_B, exchange the signatures of the respective state (Alice signs State^1_B, Bob signs State^1_B), and then exchange the revocation secrets of the previous states r^0_A and r^0_B. If Bob now tries to cheat by posting State^0_B because he holds more coins than in State^1_B, there is a spending condition on his balance, and he cannot spend it right away. Alice has one day to use r^0_B to punish Bob and steal all his money. If Bob had posted the latest state State^1_B, Alice would not have known r^1_B yet and could not have punished Bob. This ensures that honest parties cannot lose their funds, but comes at the cost of users constantly needing to monitor the blockchain to react to misbehavior, having two versions of each state, and needing a punishment mechanism per output.

enforcing the latest state (State channels, Eltoo). This is in contrast to sending funds on the blockchain itself, where users can go safely offline. To overcome this drawback, users can outsource this monitoring to a third party, a *watchtower* [Unl, KNW19, ALS+18, MBB+19a, ATLW20, MSYS21], which is either trusted or needs to lock up collateral for every channel that is watched over, which does not scale well. Therefore, we can ask the following question. *Can we get rid of the online assumption in bidirectional payment channels that support an unlimited number of payments without employing watchtowers?*

Limitation L2 (Generalizing Channel Functionality). Another fundamental gap is that there exist essentially two categories of channel constructions: (i) application-specific channels (e.g., payment channels), which offer a functionality that is specifically tailored to one application (e.g., payments) and is strictly weaker than the scripting capabilities of the underlying blockchain; and (ii) state channels, which offer the functionality for Turing-complete smart contracts, but are in turn only compatible with blockchains supporting Turing-complete scripting. We propose the following research question. *Can we construct channels that are application-agnostic while not relying on Turing-complete scripting, yet lifting the given scripting capabilities of a blockchain (layer-1) to the off-chain channel setting (layer-2)?*

1.2.2 State of the Art and Limitations in Payment Channel Networks

To achieve MHPs, two approaches exist: (i) protocols that proceed in one round of communication but lack security guarantees and (ii) protocols that proceed in two or more rounds. The second round ensures that honest users are not at risk of losing their funds but also increases communication and, thereby, the chances for errors, e.g., due to offline users. By round of communication, we mean pair-wise, sequential communication from sender to receiver (or vice-versa). Since one-round protocols put user funds at risk, two-round protocols are used in practice.

The most prominent protocol is based on HTLCs, i.e., *Hash Time Locked Contracts* or sometimes *Hashed TimeLock Contracts* (see Figure 1.5), as for instance, implemented in the Lightning Network [PD16]. On a high level, each channel on the path locks the payment amount as collateral in an HTLC (round 1), which is a simple smart contract that gives the money to the user closer to the receiver (right user) if she knows a pre-image of a hash, or else to the user closer to the sender (left user) after some time expires. Then, this pre-image is passed from right to left (round 2), resolving the HTLCs and finalizing the payment. We illustrate this in Figure 1.6.

To give users enough time to propagate the pre-image, the timelocks on these HTLCs need to grow from right to left. This means that asymptotically, the timelocks on the collateral (*collateral lock time*) are linear in the path length, something that is undesirable because it incurs high opportunity costs. [MBB+19b] is an MHP construction that has constant collateral lock time but requires Turing-complete smart contracts. [JLT21] is Bitcoin-compatible and has a logarithmic collateral lock time.

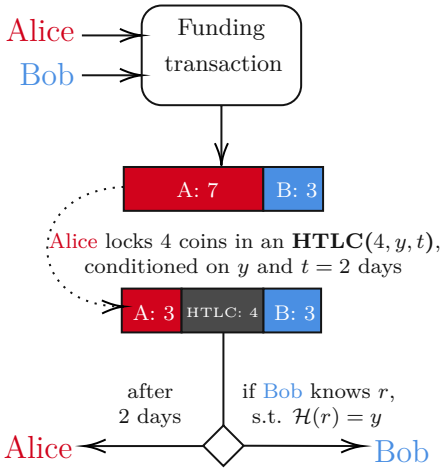

Figure 1.5: Hash Time Locked Contract (HTLC): This example shows an HTLC inside the channel of Alice and Bob. Here, Alice and Bob create and update their payment channel to a new state, where Alice locks 4 of her coins in an HTLC, conditioned on y and $t = 2$days. These 4 coins can be spent either by Bob, if Bob knows a pre-image r, such that $\mathcal{H}(r) = y$, or else by Alice, after 2days.

Additionally, all HTLCs are conditioned on the same hash, which is based on a pre-image chosen by the receiver, which in turn leads to privacy and security issues. [MMSK+17] and [MMS+19] try to address these issues by re-randomizing this secret (on a high level).

Limitation L3 (Secure Payments in Single Round). Up to this point, it was unknown whether it is possible to consolidate the communication-efficient and less error-prone (due to offline users) one-round MHPs with the security of two or more round MHPs. Ideally, such a construction also has a constant collateral lock time. Therefore, we pose the following question. *Is it possible to perform secure multi-hop payments in a single round of communication?*

Limitation L4 (Multi-Channel Updates). MHPs can only be used to update channels on a path. However, some applications, e.g., crowdfunding, mass payments, or transaction aggregation [TYA+22], require atomically updating arbitrary channels (i.e., not forming a path). [EMSM19] tries to address this issue but falls short as it is vulnerable to *channel closure attacks*, as pointed out in [JLT21], which can result in honest users losing their funds. So far, no construction has achieved this securely. Therefore, we ask the following research question. *Can we atomically update multiple channels that are not necessarily aligned on a path?*

1.2.3 State of the Art and Limitations in Virtual Channels

Virtual channels (VCs) were first introduced in [DEFM19] for Ethereum-like cryptocurrencies, i.e., cryptocurrencies that support Turing-complete smart contracts. With this construction, users can construct a VC over one intermediary user on top of two state

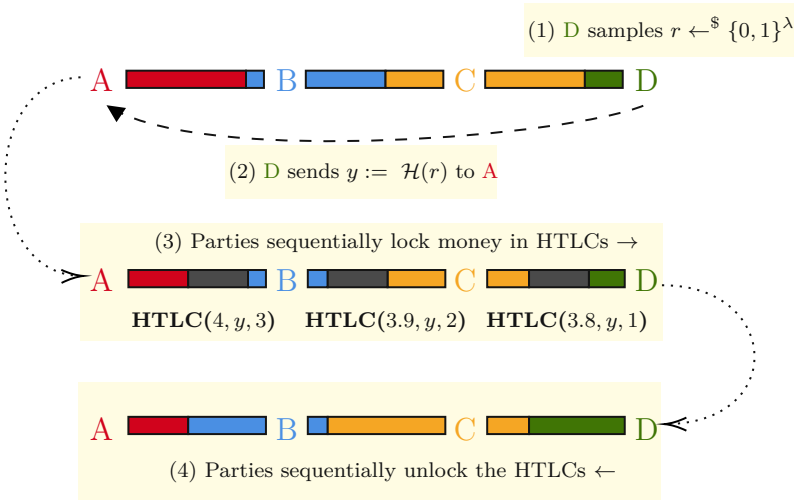

Figure 1.6: Lightning payment: Sender A pays 4 coins to receiver D via B and C. The colored boxes connecting two users represent payment channels (as shown in Figures 1.1 and 1.5). The first row shows the initial balance distribution of the channels. The last row shows the desired outcome, where 4 coins were transferred to the receiver. These updates should occur atomically. In the case of Lightning-based HTLC payments, this is achieved in four steps. (1) D samples a uniformly random string r, and (2) sends its hash $y := \mathcal{H}(r)$ to A. Then, in step (3), parties lock 4 coins in an HTLC, sequentially from left to right. B and C each charge 0.1 fee in this example, thus forward only 3.8 and 3.9 coins, respectively. In step (4), after creating all HTLCs, D knowing r can unlock his HTLC with C, claiming the coins and revealing r to C, who can continue in the same way until all HTLCs are unlocked. Due to the increasing timelocks (1, 2, and 3 days, respectively), each user has enough time to propagate r. Users are incentivized to unlock the HTLC. If something goes wrong before step (3) is completed, or if D chooses not to reveal r, the HTLCs are reverted after the timelocks expire. Thus, the payment is atomic.

channels. Later, in [DFH18], they were refined to support multiple intermediaries and in [DEF+19b] to support multi-party channels.

Limitation L5 (Bitcoin-Compatible Virtual Channels). Designing VCs in a secure way is challenging. They involve three (or more) parties, which are possibly malicious or even colluding. In the aforementioned Ethereum-based constructions [DEFM19, DFH18, DEF+19b], Turing-complete smart contracts take care of conflict resolution in case of a dispute on what is the latest state of the VC. Up to this point, it was unknown if VCs could be constructed on blockchains with more limited scripting capabilities, e.g., Bitcoin. Thus, the state of the art begs the following research question. *Is it possible to construct virtual channels in Bitcoin?*

Limitation L6 (Domino Attack). It turns out that there are Bitcoin-compatible

VC constructions, as shown by our work [AME+21] and concurrently by [JLT20]. While [AME+21] provides two constructions for VCs over one intermediary, [JLT20] provides a construction over multiple intermediaries but is built on payment channels with limited lifetime and thus not compatible with Lightning Network channels. [KL] introduces a more efficient construction, but requires an op_code not supported in Bitcoin. As it turns out, these VC constructions share a common design paradigm, which unfortunately leads to a severe new attack, which we name *Domino attack*. This attack allows an attacker to shut down the underlying PCN. Given this state of the art, we propose the following research question. *Can we construct Bitcoin-compatible virtual channels over multiple hops and be secure against the Domino attack?*

1.3 Methodology

These issues pose a unique set of challenges; to overcome them, we need to design novel, distributed protocols that combine cryptographic primitives with on-chain adjudication provided by the scripting enabled by the underlying cryptocurrency. Due to our focus on limited scripting capabilities, we cannot rely on Turing-complete smart contracts, which execute complex logic and checks but are instead less expressive and can only perform simpler checks. Thus, we need to ensure a majority of our logic in some other way, e.g., cryptographically with adaptor signatures.

Applications built on top of blockchains are inherently critical as they are related to money. To deal with this, a formal approach is necessary. Therefore, we define security and privacy properties for our protocols, formalize the protocols, and rigorously prove that the protocols achieve these properties.

Formalizing complex protocols is not trivial, especially since, in our case, the mutually untrusted protocol parties not only interact with themselves but with other protocols, e.g., blockchain or payment channel protocols. One approach to formalizing protocols in such a composable way is the Universal Composability (UC) framework [Can01, CDPW07], which we use for our protocols [AMSKM21, AMSKM23, AME+21, AEE+21, ATM+22, AAM22, SAAM23]. This framework allows us to formalize protocols in a way such that security and privacy properties are preserved when composed with other protocols, which is known as universal composition.

Applying the UC framework consists of defining an ideal functionality that acts as a trusted third party and defines the idealized behavior of the protocol, as well as a real-world protocol representing the actual protocol. The objective is then to show that the real-world protocol and the ideal protocol based on the ideal functionality are computationally indistinguishable to an outside environment. This means any attack that can be carried out in the real world is likewise possible in the ideal world. This is accomplished by defining a simulator that converts any attack on the real-world protocol into an attack on the ideal functionality. Employing the UC framework requires formal definitions of both the real-world protocol and the ideal functionality and demonstrating their computational indistinguishability through the use of said simulator.

Complementing this approach, we also describe an attack against existing protocols [AMSKM23], where we ensure to (i) clearly demonstrate the attack process and (ii) determine if it constitutes an attack on the model itself or a flaw in the proof. In the case of the attack we find in [AMSKM23], it is an attack on the model. We rely on game-based proofs to formalize and prove adaptor signatures in [AEE+21].

Experimental Evaluation and Simulations. To show the practical viability of our work, we try to incorporate evaluations and simulations wherever appropriate. For example, for protocols, we generally develop open-source proof-of-concept prototypes to demonstrate compatibility with (for instance) Bitcoin and to measure and compare the cost in terms of computation, communication, and fees against existing protocols. For attacks, algorithms, or to analyze how a protocol might perform in practice [AMSKM21, AMSKM23], we simulate the impact on a real-world data snapshot. We provide these tools on GitHub, along with the data they operate on, to ensure reproducibility.

1.4 Contributions

1.4.1 Sleepy Channels: Bi-directional Payment Channels without Watchtowers

In Chapter 2, which presents [ATM+22], we address L1. We provide a protocol that gets rid of the online assumption that is present in existing bidirectional payment channel constructions with unlimited payments. Instead of having to be online constantly (e.g., at least once per day in Lightning) and watching out for misbehavior, parties can safely go offline for extended periods of time and come online only once at a pre-scheduled time.

Additionally, we eliminate the requirement for relative timelocks, which means that the scheme is compatible with a wide range of cryptocurrencies. Theoretically, the necessary absolute timelocks could be replaced with timelock puzzles [RSW96], making the scheme compatible with any blockchain that supports digital signatures.

We evaluate the performance of this protocol with a proof-of-concept implementation. Further, we analyze the security of the scheme in the UC framework.

1.4.2 Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures

In Chapter 3, which presents [AEE+21], we address L2. We present a novel and general protocol for channels that is compatible with Bitcoin and allows to host any application that can also be encoded with the underlying scripting language. In other words, anything that can be hosted on layer-1 in Bitcoin can be deployed on layer-2 with these new channels.

We further improve efficiency (i) in terms of on-chain overhead in case of a dispute (from linear in the number of applications to constant), (ii) in terms of nested applications (from

exponential to linear in the number of recursions), and (iii) in terms of off-chain storage overhead. We achieve this by removing the need for state duplication (cf. Figure 1.4).

To still be able to identify who published the state without relying on a duplicate state, we employ adaptor signatures. Poelstra has recently introduced this cryptographic primitive in a mailing list [Poe], but we formalize it for the first time.

We further prove the security of this protocol in the UC framework and evaluate the protocol's performance and compatibility with a proof-of-concept implementation.

1.4.3 Blitz: Secure Multi-Hop Payments Without Two-Phase Commits

In Chapter 4, which presents [AMSKM21], we address L3. For the first time, we present a secure multi-hop update protocol that proceeds in one round of communication. Instead of propagating a secret like in traditional multi-hop payment schemes such as HTLC-based payments or [MMS+19], we synchronize the channels with a transaction that acts as a global event. In addition to ensuring that honest users do not lose their funds, the scheme is also secure against the wormhole attack [MMS+19].

We implement a proof-of-concept implementation to evaluate the protocol's performance and showcase its compatibility with Bitcoin and the Lightning Network. Additionally, we analyze the security and privacy of the scheme in the UC framework.

1.4.4 Thora: Atomic and Privacy-Preserving Multi-Channel Updates

In Chapter 5, which presents [AAM22], we address L4. We provide a protocol for atomically updating multiple channels on any topology, in particular also ones that are not a path.

In contrast to [EMSM19], we provide atomicity and value privacy. Further, our construction does not rely on Turing-complete smart contract capabilities, like [MBB+19b]. To this end, we enable for the first time applications that rely on non-path atomic updates of channels, e.g., crowdfunding applications.

To analyze the security and privacy, we model our protocol in the UC framework. To show compatibility with Bitcoin and evaluate the on- and off-chain overhead, we implement a proof of concept.

1.4.5 Bitcoin-Compatible Virtual Channels

In Chapter 6, which presents [AME+21], we address L5. We provide a construction that enables virtual channels in Bitcoin and other cryptocurrencies with limited scripting capabilities for the first time.

We present two constructions: virtual channels with validity (limited lifetime) and without validity (unlimited lifetime). Both constructions work via one intermediary and have some slightly different properties. We formally model these constructions in the UC framework to analyze security.

We implement a proof-of-concept implementation that shows the compatibility with Bitcoin and Lightning Network channels. Further, we conduct an evaluation measuring the on-chain and off-chain overhead.

1.4.6 Breaking and Fixing Virtual Channels: Domino Attack and Donner

In Chapter 7, which presents [AMSKM23], we address L6. We analyze existing Bitcoin-compatible virtual channel constructions [AME+21, KL, JLT20], find a new attack on these constructions, and introduce a novel construction that is secure against it.

In particular, we identify a common design in these constructions: The virtual channel is funded from the underlying payment channels. The novel attack, which we name *Domino attack*, is a direct consequence of this design. The attack is so severe that it allows an adversary to shut down the underlying PCN itself. We conduct a simulation of this attack on the LN to show the damage that an attack could cause. We also discuss other shortcomings, such as only supporting a single intermediary, latency, linear overhead on the blockchain, non-constant per-user storage overhead, and unfair fee models.

We then introduce Donner, a new virtual channel construction that uses a different design. Donner not only overcomes these drawbacks but also provides security against the Domino attack. To analyze the security and privacy of Donner, we formalize the construction in the UC framework.

We implement a proof-of-concept implementation that shows the compatibility of Donner with Bitcoin and measures its on-chain costs. Donner is also compatible with Lightning Network payment channels.

CHAPTER 2

Sleepy Channels: Bi-directional Payment Channels without Watchtowers

Abstract

Payment channels (PC) are a promising solution to the scalability issue of cryptocurrencies, allowing users to perform the bulk of the transactions off-chain without needing to post everything on the blockchain. Many PC proposals, however, suffer from a severe limitation: Both parties need to constantly monitor the blockchain to ensure that the other party did not post an *outdated* transaction. If this event happens, the honest party needs to react promptly and engage in a *punishment* procedure. This means that prolonged absence periods (e.g., a power outage) may be exploited by malicious users. As a mitigation, the community has introduced *watchtowers*, a third-party monitoring the blockchain on behalf of off-line users. Unfortunately, watchtowers are either trusted, which is critical from a security perspective, or they have to lock a certain amount of coins, called collateral, for each monitored PC in order to be held accountable, which is financially infeasible for a large network.

We present *Sleepy Channels*, the first bi-directional PC protocol without watchtowers (or any other third party) that supports an unbounded number of payments and does not require parties to be persistently online. The key idea is to confine the period in which PC updates can be validated on-chain to a short, pre-determined time window, which is when the PC parties have to be online. This behavior is incentivized by letting the parties lock a collateral in the PC, which can be adjusted depending on their mutual trust and which they get back much sooner if they are online during this time window. Our protocol is compatible with any blockchain that is capable of verifying digital signatures

(e.g., Bitcoin), as shown by our proof of concept. Moreover, our experimental results show that Sleepy Channels impose a communication and computation overhead similar to state-of-the-art PC protocols while removing the watchtower's collateral and fees for the monitoring service.

This chapter presents the results of a collaboration with Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei, which was published at the ACM Conference on Computer and Communications Security (CCS) in 2022 under the title "Sleepy Channels: Bi-directional Payment Channels without Watchtowers". Sri AravindaKrishnan Thyagarajan and I contributed equally to this work and are considered to be co-first authors. I contributed to the design of the construction, which was a joint effort with the other co-authors. Sri AravindaKrishnan Thyagarajan is responsible for writing the protocol and the comparison to related work. I am responsible for the formalization of the protocol, the ideal functionality, the implementation of a proof-of-concept, evaluation, and conducting the experiments, as well as the proof. Pedro Moreno-Sanchez, Giulio Malavolta, and Matteo Maffei were the general advisors and contributed with continuous feedback.

2.1 Introduction

Bitcoin has put forward an innovative payment paradigm both from the technical and the economical point of view. A permissionless and decentralized consensus protocol is leveraged to agree on the validity of the transactions that are afterwards added to an immutable ledger. This approach, however, severely restricts the transaction throughput of decentralized cryptocurrencies. For instance, Bitcoin supports about 10 transactions per second and requires confirmation times of up to 1 hour.

Payment channels (PC) [PD16] have emerged as one of the most promising scalability solutions. A PC enables an arbitrary number of payments between users while only two transactions are required on-chain. The most prominent example, currently deployed in Bitcoin, is the Lightning Network (LN) [Liga], which at the time of writing hosts bitcoins worth more than $130M$ USD, in a total of more than 19k nodes and more than 81k channels.

In a bit more detail, a PC between Alice and Bob is created with a single on-chain transaction *open-channel*, where users lock some of the coins into a shared output controlled by both users (e.g., requiring a 2-of-2 multisignature), effectively depositing their coins and creating the channel. Both users additionally make sure that they can get their coins back at a mutually agreed expiration time. After the channel has been successfully opened, they can pay each other arbitrarily many times by exchanging authenticated off-chain messages representing updates of their share of coins in the shared output. The PC can be finally closed by including a *close-channel* transaction on-chain that effectively submits the last authenticated distribution of coins to the blockchain (or after the PC has expired).

Issue with bidirectional channels. While the initial versions of payment channels were unidirectional (i.e., only payments from Alice to Bob were allowed), several designs for bi-directional payment channels have been proposed so far. The technical crux of these protocols is to ensure that no coins are stolen between the mutually untrusted Alice and Bob. To illustrate the problem, imagine that the current balance of the channel *bal* is {Alice:10, Bob:5 }. Alice pays 3 coins to Bob, moving the channel balance to *bal'* as {Alice:7, Bob:8}. At this point, Alice benefits from *bal* while Bob would benefit if *bal'* is the one established on-chain.

The different designs of bi-directional payment channels available so far provide alternative solutions for this crucial dispute problem (see Table 2.1). One approach consists on leveraging the existence of Trusted Execution Environment (TEE) at both Alice and Bob [LEPS16]. This approach, however, adds a trust assumption that goes against the decentralization philosophy of cryptocurrencies and it is unclear whether it holds in practice [CCX+19, VBOM+19]. Another approach consists on relying on a third-party committee [AKKWZ21, CCF+21] to agree on the last balance accepted by Alice and Bob. Again, this adds an additional assumption on the committee, and current proposals work only over smart contracts as those available in Ethereum.

The most promising approach in terms of reduced trust assumptions and backwards compatibility with Bitcoin, which is the one implemented in the LN, is based on the encoding of a punishment mechanism that allows Alice (or Bob) rescue all the coins in a channel if Bob (or Alice) attempts to establish a *stale* or *outdated* balance on-chain. Following with the running example, after the balance bal' is established, Alice and Bob exchange with each other a revocation key associated to *bal* that effectively allows one of the parties to get all the coins from *bal* if it is published on-chain by the other party.

In detail, imagine that after bal' has been agreed and *bal* has been revoked, Alice (the case with Bob is symmetric) attempts to close the channel with balance *bal*. As soon as *bal* is added on-chain, a small punishment time δ is established within which Bob can transfer all coins in *bal* to himself with the corresponding revocation key. After δ has expired, *bal* is established as final. This mechanism with time δ is called *relative timelock*[1] in the blockchain folklore (i.e., relative to the time *bal* is published).

The reader might ask at this point: And what happens if Bob does not monitor the blockchain on time (e.g., Bob crashes or he is offline) to punish the publishing of *bal*? In that case, Alice effectively manages to publish an old state that would be more beneficial for her. Therefore, the above mechanism makes an important requirement for the channel users: Both Alice and Bob have to be online persistently to ensure that if one of them cheats, the other can punish within δ. However, if Alice and Bob are regular users, it is highly likely that they go offline sporadically if not for prolonged periods of time. Moreover, existing currencies like Monero do not possess the capability for *relative timelock* in their script, and therefore the approach falls short of backwards compatibility with some prominent currencies.

[1]This can be realized via `checkSequenceVerify` (CSV) script available in Bitcoin.

The role of watchtowers. In order to avoid this problem, honest users (Bob in our running example) can rely on a third party, called *Watchtower*, that does the punishing job on his behalf. Several watchtower constructions have been proposed so far [Unl, ATLW20, KNW19, MSYS21, MBB+19a, ALS+18], but they all share the same fundamental limitation: watchtowers are either trusted, which is critical from a security perspective, or they have to lock a certain amount of coins, called collateral, for each monitored channel in order to be held accountable, which is financially infeasible for a large network.

Given this state of affairs, in this work, we investigate the following question: *Is it possible to design a secure, and practical payment channel protocol that does not require channel parties to be persistently online, nor additional parties (not even watchtowers) or additional trust assumptions, and is backwards compatible (no complex scripts) with current UTXO-based cryptocurrencies?*

Table 2.1: Comparison among payment channel approaches. We do not consider [AKKWZ21, CCF+21] as they rely on third-party committees with additional trust assumptions. Online assumption refers to the honest user being online for revocation of an old state on-chain. Unrestricted lifetime means the protocol does not require users to close the channel before a pre-specified time. Unbounded payments refer to channel users making any number of payments while the channel is open. In terms of scripts, DS refers to digital signatures, SIGHASH_NOINPUT refers to a specific signature scheme [DRO], Seq. number refers to attaching a state number to a transaction and verifying if it is greater or smaller than the current height of the blockchain. In the case of Duplex [DW15], d is the number of payments made in the channel. LRS refers to the Linkable Ring Signature scheme used in Monero [TMSS22], and DLSAG refers to the transaction scheme proposed in [MSBL+20].

	Bi-directional	Pre-schedule online	Unrestricted lifetime	Unbounded payments	Script requirements[1]
Spillman [Spi]	✗	✓	✗	✓	DS
CLTV [Tod]	✗	✓	✗	✓	DS + CLTV
Duplex [DW15]	✓	✓[2]	✗	✗	DS + CLTV
Eltoo [DRO]	✓	✗	✓	✓	DS + CSV + SIGHASH_NOINPUT + Seq number
Lightning [Liga]	✓	✗	✓	✓	DS + CSV
Generalized [AEE+21]	✓	✗	✓	✓	DS[3] + CSV
Paymo [TMSS22]	✗	✓	✗	✓	Monero's LRS + CLTV
DLSAG [MSBL+20]	✗	✓	✗	✓	DLSAG + CLTV
Teechan [LEPS16]	✓	✓	✓	✓	DS + TEE
This work	✓	✓	✗	✓	DS + CLTV
This work+ [TBM+20]	✓	✓	✗	✓	DS

[1]: Requiring less script capabilities from the blockchain results in better compatibility with currencies, and better on-chain privacy (fungibility).

[2]: This requires that the transactions of the first level of the tree use CLTV instead of CSV.

[3]: The digital signature scheme used must have adaptor signature [AEE+21] capability.

2.1.1 Our Contribution

In this work, we answer this question in the affirmative. We design *Sleepy Channels*, a new bi-directional payment channel protocol (Section 2.5) that does not require either of the channel parties to be persistently online, and therefore does not require the services of a watchtower. Our protocol allows users to schedule ahead of time when they have to come online to validate possible channel updates. This requirement is present even in

the watchtower proposals [Unl, ATLW20, KNW19, MSYS21, MBB+19a, ALS+18], where the users are required to come online before a specific time to ensure the watchtower has acted correctly. Moreover, our protocol does not make use of any complex script and is therefore backwards compatible with existing UTXO-based cryptocurrencies, many of which can avail bi-directional payment channels without additional trust assumptions for the very first time.

At the core of our Sleepy Channels protocol, we have a novel collateral technique that plays a dual role: (1) Enables the punishment of a misbehaving channel user within a predetermined time, irrespective of when the cheating exactly takes place. In technical terms, we no longer require *relative timelocks* (CSV). (2) Incentivises a channel user to cooperate in closing the channel if the other channel user wishes to do so. Our collateral technique requires both users to lock some amount of collateral each (same or different amounts for the two users), whose exact value is determined by the level of trust between the users: A high trust level means low collateral, while a low trust level means high collateral.

Our protocol only involves signature generation on mutually agreed transactions, along with the use of *verifiable timed signatures* [TBM+20, TMSS22] for achieving backward compatibility with existing currencies, especially privacy-preserving currencies like Monero for the *first time*. With the aid of techniques from [TBM+20, TMSS22], the transactions in our protocol look exactly the same as any other regular transaction in the currency, thereby ensuring high *fungibility*. If the currency already supports `checkLockTimeVerify` (CLTV) script[2], then our protocol only requires signature generation.

We formally prove the security of our Sleepy Channels protocol in the *Universal Composability (UC)* [Can00] framework. For this, we design an ideal functionality (in Section 2.4) that captures a bi-directional payment channel with the same security and efficiency guarantees as the functionality from [AEE+21], except that we achieve *delayed finality with punish*. This notion guarantees that until some time **T**, an honest party can receive coins according to either the latest payment state or all the coins from the channel (if the other misbehaves). Due to space constraints, the formal protocol description and the security analysis in the UC framework can be found in Appendix A.1.

We evaluate the performance of our Sleepy Channels protocol in the presence of CLTV and our results show that the time and communication cost are in line with the highly efficient protocols used in Lightning Network (LN) [Liga]. We further conduct two simulation experiments. In the first, we measure how much centralized collateral watchtower service providers need to allocate, in order to serve a certain percentage of the LN. We analyze watchtower proposals that fully collateralize the channels, e.g., [ATLW20, MSYS21, MBB+19a]. For 30% of the LN, this amounts to around 890 BTC (or roughly 39M USD) of collateral. For Sleepy Channels, on the other hand, the collateral is distributed, without the need of a central entity owning this amount of money. In the second experiment,

[2]The script (available in Bitcoin) sets a transaction to be valid only after some pre-specified height (t) of the blockchain. That is, the transaction is set to be valid only after some point in time in the future.

we measure the channels at risk of having their funds stolen given a chance of failing to come online once a day over a given time period. Using LN channels over a one-month period, for a chance of 0.1% there are 5k channels at risk, for a chance of 1% there are 49k channels at risk (roughly 60% of LN). For Sleepy Channels over the same period, there are around 97% fewer channels at risk.

2.1.2 Related Work

Below, we discuss and compare other prior works that are relevant to our work.

Comparison to other payment channel protocols. CLTV [Tod] and Spillman [Spi] proposed uni-directional payment channels between Alice and Bob where payments could only be made to Bob and thus the balance of Bob only increases. Therefore there was no payment revocation as Bob always preferred the most recent payment. Moreover, the channel had a fixed expiry that is set at the time of the channel creation. Duplex channels [DW15] support bi-directional channels but only support a limited number of payments as with each successive payment, the lifetime of the channel decreases. Moreover, the protocol requires $\log d$ number of transactions to close the channel where d is the number of payments made. Other payment channel proposals typically require only one transaction to close. Eltoo [DRO] also supports bi-directional payments but requires a special signature scheme like SIGHASH_NOINPUT, relative timelocks (CSV), and related scripts, and therefore is not compatible with several of the existing currencies, including Bitcoin itself. Lightning channels [Liga] are the most popular channels currently in use that support bi-directional payments but require relative timelocks (CSV). Generalized channels [AEE+21] support bi-directional payments but again require relative timelocks (CSV). More importantly, they require the underlying signature scheme to support adaptor signatures [AEE+21] capability[3]. Paymo [TMSS22] and DLSAG [MSBL+20] are proposals tailored for Monero that only support uni-directional payments. Teechan [LEPS16] is a bi-directional payment channel proposal but requires both users to possess TEEs. A summary of the comparison is presented in Table 2.1.

Payment channels that support arbitrary conditional payments are referred to as *state channels* [DFH18, DEFM19, CCF+21] and require complex scripts like *smart contracts* and are incompatible with UTXO-based currencies. Bi-directional payments can also be realized by making use of the smart contract support of a third ledger (like Ethereum) via *ZK-Rollups* [ZK-]. However, the solutions available are far from ideal either due to high computational costs off the chain or high costs on-chain in terms of gas costs or transaction fees [Cry]. Moreover, zk-rollups rely on a coordinator for liveness, meaning that if the coordinator goes offline, every user must submit a punishment transaction on-chain, which is costly, effectively closes all channels and largely increases the overhead on-chain. Finally, such a coordinator is in the position to observe every single transaction

[3]Recently it was shown that deterministic signatures do not possess adaptor signature capabilities [EFH+21], that includes signature schemes like BLS.

between any two users (thus largely limiting their privacy) and decide whether to process such transactions or censor them instead.

Additionally, the payment channel proposals can be compared based on the number of transactions it requires to close a channel. For comparable security, we consider the prior payment channel protocols to be supported by the state-of-the-art watchtower proposal [MSYS21]. We have that when parties are honest and trustful of each other, prior works require 2 transactions to close a channel (one to close the channel and one for watchtower collateral), while Sleepy Channels require only 1 transaction. In the case where parties are honest and distrustful, prior works require in total 3 transactions to close the channel same as Sleepy Channels if parties wish for a fast closure. A notable exception is Duplex which requires log d. In case where parties are dishonest and in the worst case, prior works like Duplex require $\log d$, Eltoo requires 5, Lightning requires 4, and generalized channels requires 5 transactions in total. Sleepy channels on the other hand require only 3 transactions in total. Here total refers to the total number of transactions to misbehave, punish, and close the channel.

Advantages over watchtowers. As discussed above, parties may avail the services of a third party like a *watchtower*. Monitor [Unl] is a watchtower proposal requiring no special scripts. However, an offline watchtower is not penalized and may even get rewarded if a revoked payment is successful on-chain. DCWC [ALS+18] is another such proposal that fails to penalize an offline watchtower where the honest user ends up losing coins as a revoked payment is posted on the chain. Outpost [KNW19] requires an OP_RETURN script and also requires the channel user (hiring the watchtower) to pay the watchtower for every channel update. The OP_RETURN script (available in Bitcoin) is used to enter arbitrary information of limited size into a transaction. This however increases the size of the transaction thus requiring a transaction higher fee, and also affects the fungibility of the coins involved in the transaction. PISA [MBB+19a] heavily relies on smart contract support and also requires the watchtower to lock large collateral (equal to the channel capacity) along with the channel. Cerberus channels [ATLW20] and FPPW [MSYS21] are recent proposals that suffer from the problem of revealing the channel balance to the watchtower per update and therefore lack balance privacy. Similar to PISA, they also require the watchtower to lock large collateral along with the channel.

All of the above watchtower proposals also fundamentally lack *channel unlinkability* as the watchtower can clearly track channel-related transactions on-chain. Except for PISA, all of the above proposals still require relative timelocks (CSV), which can be replaced with absolute timelocks (CLTV) at the expense of restricted lifetimes for the channels. To incentivize watchtowers, the above protocols require the users to pay a one-time or a persistent fee to the watchtower *even* if the users behave honestly. On the other hand, users of Sleepy Channels do not lose any coins under honest behaviour as they are guaranteed to get back their collateral.

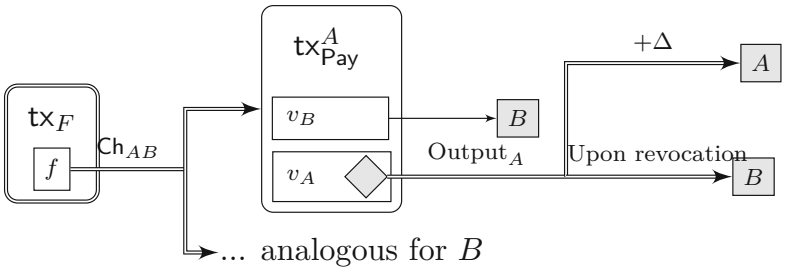

Figure 2.1: The transaction flow of LN channel between A and B. Rounded boxes represent transactions, rectangles within represent outputs of the transaction: here $v_A + v_B = f$. Incoming arrows represent transaction inputs, while outgoing arrows represent how an output can be spent. Double lines from transaction outputs indicate the output is a shared address. A single line from the transaction output indicates that the output is a single party address. We write the timelock (Δ) associated with a transaction over the corresponding arrow.

2.2 Solution Overview

In this section, we give a high-level overview of our construction. We start by reviewing the state of the art in payment channels, i.e., those employed in the Lightning Network (LN) [Liga], illustrating its limitations, and, based on that, gradually introducing our solution. Our solution consists of a base solution that removes the need for users to constantly be online and an optional extension that aims to disincentivize users from blocking funds in the case that they are online.

Lightning channels. Two parties A and B lock up some money in a joint address (or channel) Ch_{AB}, as described in Figure 2.1. They can perform payments to each other by exchanging payment transactions $\mathsf{tx}_{\mathsf{Pay}}$, which commit to an updated balance of both users, v_A for A and v_B for B in this case. Party A gets a signed transaction $\mathsf{tx}^A_{\mathsf{Pay}}$, while party B gets a signed transaction $\mathsf{tx}^B_{\mathsf{Pay}}$, both of which reflect the above payment state. In order for this mechanism to be secure, the parties need to revoke the previous state whenever an update is performed. This is done by exchanging a punishment transaction that gives the balance of the cheating user to the honest user, should the former try to post an old (revoked) state. To give precedence to the punishment transaction, if party A posts $\mathsf{tx}^A_{\mathsf{Pay}}$, it is forced to wait for a relative timelock of Δ (in practice, one day) until it can spend the balance v_A, in order to give time to the other party B to punish. Notice that party B can spend its balance v_B immediately after $\mathsf{tx}^A_{\mathsf{Pay}}$ is posted. On the other hand, we have the analogous case for party B with the transaction $\mathsf{tx}^B_{\mathsf{Pay}}$.

With this mechanism in place, a party that wants to prevent being cheated on needs to be online constantly throughout the lifetime of the channel and monitor the blockchain for old states. If it does, it has Δ time units *immediately after* the posting of $\mathsf{tx}_{\mathsf{Pay}}$ to perform the punishment. One workaround for this problem is to employ a trusted third party, a *Watchtower*, which takes over the responsibility of monitoring the ledger, thereby allowing a party to safely go offline. As pointed out previously, this approach has

fundamental drawbacks such as the need for the Watchtower to lock up coins for each channel that it watches over, besides the fees requested by the Watchtower for its service.

Attempt to remove relative timelock. To drop the requirement for users to constantly be online, an attempt is to replace the relative timelock of Δ time units in Figure 2.1 with an absolute timelock until time $\mathbf{T}$. This is done by specifying $\mathbf{T}$ as a block height using the CLTV script. In other words, the party A that posts a state $\mathsf{tx}^{A}_{\mathsf{Pay}}$ has to wait until time $\mathbf{T}$ (irrespective of when $\mathsf{tx}^{A}_{\mathsf{Pay}}$ is posted on the chain) before it can retrieve the funds v_A. This allows B to safely go offline during the channel lifetime and only come back shortly before $\mathbf{T}$ to check if an old state was posted by A. We note that this is completely symmetric: A can safely go offline until shortly before $\mathbf{T}$ and then check whether or not B has posted an old state $\mathsf{tx}^{B}_{\mathsf{Pay}}$. However, this naive attempt punishes honest parties that wish to close their channels. That is, in the case where an honest party, w.l.o.g. say A, posts the latest state, it still needs to wait until time $\mathbf{T}$ before having access to its funds v_A. This could be undesirable as $\mathbf{T}$ could span several weeks.

Counter-party confirmation. While it is true that B (the counter-party) can safely go offline until shortly before $\mathbf{T}$, this is of course optional and one could think of cases where B is not offline. In the case that B is online, B can go ahead and confirm that A did not misbehave, i.e., A posted the latest state. So if B is online and decides to retrieve its funds v_B (thereby implicitly confirming the state dictated by $\mathsf{tx}^{A}_{\mathsf{Pay}}$), A's funds should be automatically unlocked as well.

We can implement this improvement as shown in Figure 2.2 where the counter-party B can confirm a payment transaction thus enabling party A to immediately retrieve its funds v_A and not wait until $\mathbf{T}$. To do this, after A posts the state $\mathsf{tx}^{A}_{\mathsf{Pay}}$, B has the option (in the case B is online) to post the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$ (along with a signature on it) which lets A unlock its funds immediately by means of posting the transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay}}$ (along with a signature on it). Another way to think of this is that by unlocking B's funds using $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$, B gives a confirmation that $\mathsf{tx}^{A}_{\mathsf{Pay}}$ is indeed the latest state. On a technical level, the parties A and B would create a fast unlock transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay}}$ that can be spent if B puts its transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$, using an output thereof as input. With this improvement, A's money v_A either stays locked until $\mathbf{T}$ if B is offline or A's money becomes unlocked as soon as B spends its output v_B, in case B is online before $\mathbf{T}$.

2.2.1 Extension: Incentivizing a fast unlock

In the above solution, note that the balance v_B that B committed to in the latest state can be very small or even 0, such that the incentive for B to give this fast confirmation is small or nonexistent. This leaves A to wait for a potentially long time (until $\mathbf{T}$) and opens the door to *Denial-of-Service* (DoS) attacks from B.

To avoid a situation where B is online, but has no or little incentive to unlock its funds and thereby let A unlock its channel balance early, we add the following extension (as described in Figure 2.3). To add an incentive for B to unlock early, we let B add a

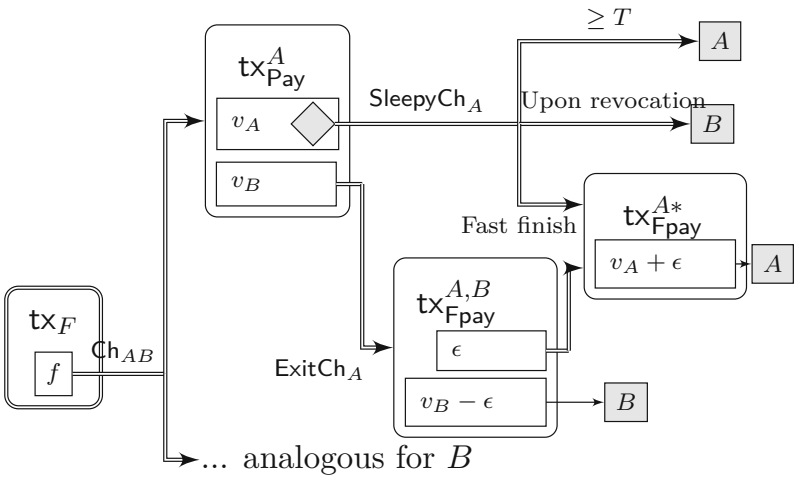

Figure 2.2: Transaction flow of our base solution. Here double lines from transaction outputs indicate that the output is a 2-party shared address between A and B. A single line from the transaction output indicates that the output is a single party address. We have $v_A + v_B = f$ and ϵ is some negligible amount of coins.

collateral of amount c. For simplicity, let c be equal to the channel capacity f. B's collateral is locked in such a way that it remains locked until B gives a fast confirmation for unlocking A's coins. Note that A's coins are now guaranteed to be smaller (or equal in the worst case) to the amount of coins B has locked. This means that a malicious B that is online and attempts to perform a DoS attack on A, ends up locking at least as many coins from itself until $\mathbf{T}$. Also note that for the case where A posts an old state, B can first punish and then immediately unlock his coins plus collateral. Analogously, A puts the same amount c as a collateral for the symmetric case. Later in Section 2.5.2, we discuss scenarios where the two parties may lock different amounts of collateral each.

Making the collateral dynamic. We further refine this solution by changing c from the total capacity of the channel f to a parameter chosen by both parties of the channel. Depending on the level of trust between the two parties, the value of c can be anything from 0 up to f. We note that setting $c = 0$ yields the base solution. Once the two parties agree on a value for c, during the funding of the channel, they can fund the channel with the total channel capacity f plus the additional collateral $2c$ (c from each party). Note that the payments are still made with the channel capacity of f and the collateral coins $2c$ are only used as an incentive for the fast closing of channels. And after the closing, both parties A and B get back their original collateral amounts of c coins each.

There is still one problem left though. Again, if the balance of B is 0 and A's balance is the capacity of the channel f, then B can lock up c coins and will lock up $c + f$ coins of A before the fast confirmation. In a final improvement, we resolve this issue by refining the transaction $\mathsf{tx}^{A}_{\mathsf{Pay}}$ so that A gets back its part of the collateral immediately. This is safe since the collateral serves merely the purpose of incentivizing the counter-party

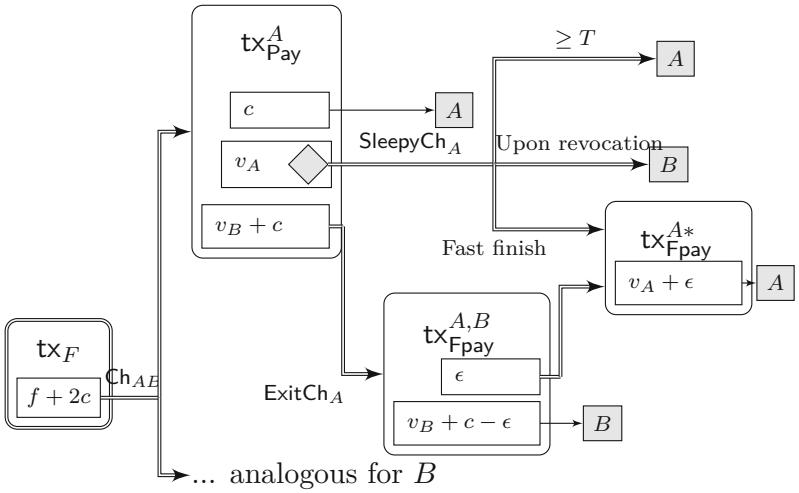

Figure 2.3: Transaction flow of the extension to our protocol. Again, $v_A + v_B = f$ and ϵ is some negligible amount of coins. The collateral c can be chosen as a value $0 \leq c \leq f$. For $c = 0$, we get Figure 2.2.

(in this case B), to acknowledge that the transaction indeed corresponds to the latest channel state. Note that the posting party A only unlocks its collateral right away and not its channel balance set by $\mathsf{tx}^{A}_{\mathsf{Pay}}$. Indeed, in the extreme case, if A posts $\mathsf{tx}^{A}_{\mathsf{Pay}}$ on the chain, A can redeem its collateral c immediately while B locks up c coins and A locks up only f coins. If $c = f$, notice that B has locked the same amount of coins as A, which discourages B from launching a DoS attack on A.

Overcoming the drawbacks. With the presented constructions (Figure 2.2 and Figure 2.3), we indeed manage to achieve bidirectional channels with unbounded payments without the need for users to constantly be online and monitor the blockchain. We offer our base solution and our extension, which puts an additional incentive on the other user to confirm states early in case they happen to be online. However, in both solutions, they can safely go offline and can come back only shortly before the pre-defined lifetime **T** of the channel. Further, our construction requires only digital signatures and absolute timelocks in the form of CLTV.

We wish to emphasize that a similar requirement of A (or B) going online shortly before **T** is present even in the watchtower proposals [Unl, ATLW20, KNW19, MSYS21, MBB+19a, ALS+18]. In that case, the hiring (channel) user Bob, is required to come online at a specific point in time **T** to check if the watchtower performed according to the protocol specification. That is, check if the watchtower indeed punished a misbehaving A correctly.

Timelock independence and compatibility. The absolute timelock in the form of CLTV makes the protocol not compatible with currencies like Monero where the CLTV script is not supported. However, the requirement of CLTV script in Sleepy Channels can be removed by making use of timed payments through *verifiable timed*

signatures (VTS) [TBM+20]. This makes Sleepy Channels applicable in a wider range of currencies as it only requires a digital signature script for cryptographic authentication from the underlying currency. For the case of Monero, making use of a variation of VTS from [TMSS22] for linkable ring signatures (instead of a standard digital signature), we can realize timed payments and thus a bi-directional payment channel in the form of Sleepy Channels for the first time. We discuss more details of the same in Section 2.5.

2.3 Preliminaries

We denote by $n \in \mathbb{N}$ the security parameter and by $x \leftarrow \mathcal{A}(\mathsf{in}; r)$ the output of the algorithm $\mathcal{A}$ on input in using $r \leftarrow \{0,1\}^*$ as its randomness. We often omit this randomness and only mention it explicitly when required. We consider *probabilistic polynomial time* (PPT) machines as efficient algorithms.

Universal composability. We model security in the *universal composability* framework with global setup [CDPW07], which lets us model concurrent executions. We consider a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ that is running the protocol. Further, we assume *static* corruptions, where the adversary $\mathcal{A}$ announces at the beginning which parties it corrupts. We denote the environment by $\mathcal{E}$, which captures anything that happens "outside the protocol execution". We model synchronous communication by using a global clock $\mathcal{F}_{clock}$ capturing execution rounds. We assume authenticated communication with guaranteed delivery between users, as in $\mathcal{F}_{GDC}$.

For a real protocol Π and an adversary $\mathcal{A}$ we write $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ to denote the ensemble corresponding to the protocol execution. For an ideal functionality $\mathcal{F}$ and an adversary $\mathcal{S}$ we write $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ to denote the distribution ensemble of the ideal world execution.

Definition 1 (Universal Composability)**.** A protocol τ UC-realizes an ideal functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{E}$ the ensembles $EXEC_{\tau,\mathcal{A},\mathcal{E}}$ and $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable.

Digital signatures. A digital signature scheme DS, lets a user authenticate a message by signing it with respect to a public key. Formally, we have a key generation algorithm $\mathsf{Gen}(1^n)$ that takes the security parameter 1^n and outputs the public/secret key pair $(\mathsf{pk}, \mathsf{sk})$, a signing algorithm $\mathsf{Sign}(\mathsf{sk}, m)$ that inputs sk and a message $m \in \{0,1\}^*$ and outputs a signature σ, and a verification algorithm $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$ that outputs 1 if σ is a valid signature on m under the public key pk, and outputs 0 otherwise. We require the standard notion unforgeability for the signature scheme [GMR88]. A stronger notion of strong unforgeability for the signature scheme was shown to be equivalent to the UC formulation of security [BH04].

2-Party computation. The aim of a secure 2-party computation (2PC) protocol is for the two participating users P_0 and P_1 to securely compute some function f over their private inputs x_0 and x_1, respectively. Apart from output correctness, we require *privacy*, i.e., the only information learned by the parties in the computation is the one determined

by the function output. Note that we require the standard *security with aborts*, where the adversary can decide whether the honest party will receive the output of the computation or not. In other words, we do not assume any form of fairness or guaranteed output delivery. For a comprehensive treatment of the formal UC definition, we refer the reader to [Can00]. In this work, we make use of 2-party signing key generation (Γ_{JKGen}) and 2-party signature generation (Γ_{Sign}) protocols [Lin21, GJKR99, BDN18].

Blockchain and transaction scheme. We assume the existence of an ideal ledger (blockchain) functionality $\mathcal{L}$ [MMSK+17, MMS+19, AEE+21] that maintains the list of coins currently associated with each address (denoted by addr) and that we model as a trusted append-only bulletin board. The corresponding ideal functionality $\mathcal{F}_{\mathbb{B}}$ maintains the ledger $\mathcal{L}$ locally and updates it according to the transactions between users. Transactions are generated by the transaction function tx: A transaction tx_A that is generated as $\mathsf{tx}_A := \mathsf{tx}([\mathsf{addr}_1, \ldots, \mathsf{addr}_n], [\mathsf{addr}'_1, \ldots, \mathsf{addr}'_m], [v_1, \ldots, v_m])$, such that it transfers all the coins (say v coins) from the source addresses $[\mathsf{addr}_1, \ldots, \mathsf{addr}_n]$ to the destination addresses $[\mathsf{addr}'_1, \ldots, \mathsf{addr}'_m]$ such that v_1 coins are sent to addr'_1, v_2 coins are sent to addr'_2 and so on, where $v_1 + v_2 + \cdots v_m = v$. Addresses are typically public keys of digital signature schemes and the transaction is authenticated with a valid signature with respect to each of the source addresses $[\mathsf{addr}_1, \ldots, \mathsf{addr}_n]$ (as the public keys). We consider *Unspent Transaction Output* (UTXO) model where an address is tied to the transaction that creates it and is spendable (used as input to a transaction) *exactly once*, like in Bitcoin, Monero, etc.

2.4 Ideal Functionality Bi-directional Channels

We define an ideal functionality $\mathcal{F}_L$ that closely follows the bi-directional payment functionality defined in [AEE+21]. In fact, our functionalitiy captures the same security and efficiency notions, except that we achieve *delayed finality with punish*, which means that the channel owner has the guarantee that until time $\mathbf{T}$, the time until which the latest state is locked, either that state or one that gives all the money to the honest party can be enforced on the ledger. Whenever one party tries to close the channel with the latest state, the other party can safely be offline until before $\mathbf{T}$, but if it stays online is incentivized to confirm it before $\mathbf{T}$, thereby unlocking not only the state but also their collateral c. We present the ideal functionality for our solution with extension and note that setting $c = 0$ yields the functionality for the base solution without collateral.

Specific notation. We abbreviate γ as an attribute tuple containing the following information $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{cash}, \gamma.\mathsf{st}, \gamma.\mathbf{T}, \gamma.c)$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the channel identifier, $\gamma.\mathsf{users}$ defines the two users of the channel, $\gamma.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ the total capacity, $\gamma.\mathsf{st}$ the list of outputs (addresses and values) in, $\gamma.\mathbf{T} \in \mathbb{R}_{\geq 0}$ defines the lifetime of the channel, and $\gamma.c \in \mathbb{R}_{\geq 0}$ the collateral of the channel.

We denote by $m \overset{\tau}{\hookrightarrow} P$ the output of message m to party P in round τ. Similarly, $m \overset{\tau}{\hookleftarrow} P$ denotes the input of message m in round τ. A message m generally consists of

(MESSAGE-ID, *parameters*). For better readability, we omit session identifiers in messages. In our communication model, messages sent between parties are received in the next round, i.e., if A sends a message to B in round τ, B will receive it in round $\tau + 1$. Messages sent to the environment, the simulator $\mathcal{S}$ or to $\mathcal{F}_L$ are received in the same round.

Description. As we do not consider privacy notions, we say that $\mathcal{F}$ implicitly forwards all messages to the simulator $\mathcal{S}$. Note that $\mathcal{F}_L$ cannot create signatures or prepare transaction ids. It expects $\mathcal{S}$ to perform these tasks, e.g., expecting a transaction of a certain structure to appear on the ledger, and outputting ERROR, if this does not happen. Similarly, whenever the functionality expects $\mathcal{S}$ to provide or set a value, but $\mathcal{S}$ does not do it, the functionality implicitly outputs ERROR, where all guarantees are potentially lost. Hence, we are interested only in protocols that realize $\mathcal{F}$, but never output ERROR.

$\mathcal{F}_L$ interacts with a ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$ parameterized over a given upper bound Δ, after which valid transactions are appended to the ledger, a signature scheme Σ and a set $\mathcal{V}$, defining valid spending conditions, including signature verification under Σ and absolute timelocks. $\mathcal{F}_L$ can see the transactions on the ledger and infer ownership of coins. Following [AEE+21], we keep the functionality $\mathcal{F}_L$ description generic, by parameterizing it over T_p and k, both of which are independent of Δ. T_p is an upper bound on the number of consecutive off-chain communication rounds between two users, while k defines the number of states that a channel has. We present a protocol later, where $k = 2$. Both T_p and Δ are defined as upper bounds. If the actual values are less, $\mathcal{S}$ implicitly informs $\mathcal{F}_L$ of these values.

The ideal functionality keeps a map Γ, which maps the id of an existing channel to the channel tuple γ representing the latest state and the address of the funding transaction, Ch_{AB}. Note that during an update, there may be two states that are active $\{\gamma, \gamma'\}$. We give a formal description of $\mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}$ (which we abbreviate as $\mathcal{F}_L$) in Figure 2.4. Following, we explain our functionality in prose and argue inline, why certain security and efficiency goals hold.

Create. When both parties of channel γ send a message $(\texttt{CREATE}, \gamma, \mathsf{tid}_P)$ to $\mathcal{F}_L$ within T_p rounds, $\mathcal{F}_L$ expects a funding transaction to appear on $\mathcal{L}$ within Δ rounds, spending both inputs tid_A and tid_B and holding $\gamma.\mathsf{cash} + 2\gamma.c$ coins. The channel funding address Ch_{AB} is stored in Γ and CREATED is sent to both parties.

Update. One party P initiates the update with $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}})$, where id refers to the channel identifier, $\overrightarrow{\theta}$ represents the new state (e.g., coin distribution or other applications that work under *delayed finality with punish*) and t_{stp} denotes the time needed to setup anything that is built on top of the channel. First, the parties agree on the new state. For this, $\mathcal{S}$ informs $\mathcal{F}_L$ of a vector of k transactions. Both parties can abort here by P not sending SETUP–OK and Q not sending UPDATE–OK. When P receives UPDATE–OK, they move on to the revocation. $\mathcal{F}_L$ expects a message REVOKE from both parties, and in the success case, UPDATED is output to both parties. In case of an error,

Ideal Functionality $\mathcal{F}_L(T_p, k)$

Create: Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}_A) \xleftarrow{\tau_0} A$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, \mathsf{tid}_B) \xleftarrow{\tau} B$, where $\tau_0 - \tau \leq T_p$: If $\mathsf{tx}_F := \mathsf{tx}([\mathsf{tid}_A, \mathsf{tid}_B], \mathsf{Ch}_{AB}, \gamma.\mathsf{cash} + 2\gamma.c)$ for some address Ch_{AB} appears on $\mathcal{L}$ in round $\tau_1 \leq \tau + \Delta + T_p$, set $\Gamma(\gamma.\mathsf{id}) := (\{\gamma\}, \mathsf{Ch}_{AB})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

Wait for B: Else wait if $(\texttt{CREATE}, \mathsf{id}) \xleftarrow{\tau \leq \tau_0 + T_p} B$ (then, "Both agreed" option is executed). If such a message is not received, stop.

Update: Upon $(\texttt{UPDATE}, \mathsf{id}, \vec{\theta}, t_{\mathsf{stp}}) \xleftarrow{\tau_0} A$, parse $(\{\gamma\}, \mathsf{Ch}_{AB}) := \Gamma(\mathsf{id})$, set $\gamma' := \gamma$, $\gamma'.\mathsf{st} := \vec{\theta}$:

1. In round $\tau_1 \leq \tau_0 + T_p$, let $\mathcal{S}$ define $\vec{\mathsf{tid}}$ s.t. $|\vec{\mathsf{tid}}| = k$. Then $(\texttt{UPDATE–REQ}, \mathsf{id}, \vec{\theta}, t_{\mathsf{stp}}, \vec{\mathsf{tid}}) \xhookrightarrow{\tau_1} B$ and $(\texttt{SETUP}, \mathsf{id}, \vec{\mathsf{tid}}) \xhookrightarrow{\tau_1} A$.
2. If $(\texttt{SETUP–OK}, \mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} A$, then $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_3 \leq \tau_2 + T_p} B$. Else stop.
3. If $(\texttt{UPDATE–OK}, \mathsf{id}) \xleftarrow{\tau_3} B$, then (if B honest or instructed by $\mathcal{S}$) send $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_4 \leq \tau_3 + T_p} A$. Else distinguish:
   - If B honest or if instructed by $\mathcal{S}$, stop *(reject)*. Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{Ch}_{AB})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
4. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_4} A$, send $(\texttt{REVOKE–REQ}, \mathsf{id}) \xhookrightarrow{\tau_5 \leq \tau_4 + T_p} B$. Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{Ch}_{AB})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
5. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_5} B$, $\Gamma(\mathsf{id}) := (\{\gamma'\}, \mathsf{Ch}_{AB})$, send $(\texttt{UPDATED}, \mathsf{id}, \vec{\theta}) \xhookrightarrow{\tau_6 \leq \tau_5 + T_p} \gamma.\mathsf{users}$ and stop *(accept)*. Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{Ch}_{AB})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.

Close: Upon $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_0} A$, distinguish

Both agreed: If already received $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau} B$, where $\tau_0 - \tau \leq T_p$, let $(\{\gamma\}, \mathsf{Ch}_{AB}) := \Gamma(\mathsf{id})$ and distinguish:

- If $\mathsf{tx}_c := \mathsf{tx}(\mathsf{Ch}_{AB}, [out_A, out_B], [\gamma.c + \gamma.\mathsf{st.balance}(A), \gamma.c + \gamma.\mathsf{st.balance}(B)])$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$, set $\Gamma(\mathsf{id}) := \bot$, send $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- Else, if at least one of the parties is not honest, run $\texttt{L–ForceClose}(\mathsf{id})$. Else, output $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop.

Wait for B: Else wait if $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau \leq \tau_0 + T_p} B$ (in that case "Both agreed" option is executed). If such a message is not received, run $\texttt{L–ForceClose}(\mathsf{id})$ in round $\tau_0 + T_p$.

Punish: (executed at the end of every round τ_0) For each $(X, \mathsf{Ch}_{AB}) \in \Gamma$ check if $\mathcal{L}$ contains a transaction $\mathsf{tx}^A_{\mathsf{Pay},i} := \mathsf{tx}(\mathsf{Ch}_{AB}, o_C, v_C)$ for some addresses o_C and some values v_C, s.t. $\sum_{v \in v_C} = \gamma.\mathsf{cash}$ and one address $o \in o_C$ belongs to A with the corresponding value $v \in v_C = \gamma.c$ for some $A \in \gamma.\mathsf{users}$ and $B \in \gamma.\mathsf{users} \setminus \{A\}$. If yes, then define $L := \{\gamma.\mathsf{st} \mid \gamma \in X\}$ and distinguish:

Punish: If B is honest and $\mathsf{tx}^A_{\mathsf{Pay},i}$ does not correspond to the most recent state in X, $\mathsf{tx}^B_{\mathsf{Pnsh},i} := \mathsf{tx}(o \in o_C, o_P, \gamma.\mathsf{st.balance}(A))$, where o_P is an address controlled by B, appears on $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$. Afterwards, in round $\tau_2 \leq \tau_1 + \Delta$ a transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i} := (o \in o_C, o_S, v_S)$, for some addresses o_S and corresponding values v_S where one address $o \in o_S$ belongs to B and the corresponding value of o is $\gamma.\mathsf{st.balance}(B) + \gamma.c - \epsilon$, appears on $\mathcal{L}$, set $\Gamma(\mathsf{id}) = \bot$, send $(\texttt{PUNISHED}, \mathsf{id}) \xhookrightarrow{\tau_2} B$ and stop.

Close: Either $\Gamma(\mathsf{id}) = \bot$ before round $\tau_0 + \Delta$ (channel was peacefully closed) or after round $\tau_1 \leq \tau_0 + \Delta$ a transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i} := (o \in o_C, o_S, v_S)$, for some addresses o_S and corresponding values v_S where one address $o \in o_S$ belongs to B and the corresponding value of o is $\gamma.\mathsf{st.balance}(B) + \gamma.c - \epsilon$, appears on $\mathcal{L}$ before a transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay},i} := ([o \in o_C, o' \in o_S], o_F, \gamma.\mathsf{st.balance}(A) + \epsilon)$ where address o_F of A appears on $\mathcal{L}$. Set $\Gamma(\mathsf{id}) := \bot$ and send $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_2 \leq \tau_1 + \Delta} \gamma.\mathsf{users}$. Else, transaction $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i} := \mathsf{tx}(o \in o_C, o_E, \gamma.\mathsf{st.balance}(A))$ where address o_E of A appears on $\mathcal{L}$ in round $\tau_3 \leq \gamma.\mathbf{T} + \Delta$. Set $\Gamma(\mathsf{id}) := \bot$ and $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_3} \gamma.\mathsf{users}$ and stop.

Error: Otherwise $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$.

Subprocedure $\texttt{L–ForceClose}(\mathsf{id})$: Let τ_0 be the current round and $(\gamma, \mathsf{tx}) := \Gamma(\mathsf{id})$. If within Δ rounds tx is still an unspent transaction on $\mathcal{L}$, then $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop. Else, latest in round $\gamma.\mathbf{T} + \Delta$, $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \texttt{ERROR}\}$ is output via Punish.

Figure 2.4: Ideal Functionality

the L–ForceClose subprocedure is executed, which expects the funding transaction of the channel to be spent within Δ rounds.

Close. Either party can initiate a channel's closure by sending $(\texttt{CLOSE}, \mathsf{id})$ to $\mathcal{F}_L$. If the other party sends the same message within T_p rounds, $\mathcal{F}_L$ expects a transaction representing the latest state of the channel to appear on the ledger within Δ rounds. Should only one party request the closure or in case one party is corrupted, $\mathcal{F}_L$ expects either a transaction representing the latest state of the channel or an older state, followed by a punishment (see Punish). If the funding transaction remains unspent, outputs ERROR.

Punish. To give honest parties the guarantee that either the most recent state of the channel which is locked until at most time $\mathbf{T}$ can be enforced on $\mathcal{L}$, or the honest party can get all coins (minus the other party's collateral), we need the punish phase. This check is executed in each round. We can model this in the UC framework, by expecting $\mathcal{E}$ to pass the execution token in every round. If $\mathcal{E}$ fails to do that, $\mathcal{F}_L$ outputs an error the next time it has the execution token. Whenever the funding transaction of any open channel γ in Γ is spent, $\mathcal{F}_L$ expects either a transaction that spends the coins in accordance to the latest state of γ, or a transaction giving $\gamma.\mathsf{cash} + \gamma.c$ coins to the honest party. Else, ERROR is output. In the case that a transaction in accordance with the latest state of γ appears on the ledger, either the funds of the party that has posted the transaction are locked until $\mathbf{T}$ (after which a transaction claiming them appears) or the other party unlocks them beforehand by unlocking their own funds and collateral. In the latter case, the other party loses the negligible amount ϵ (which we say is a system parameter in $\mathbb{R}_{\geq 0}$ for a ledger $\mathcal{L}$) to the first party.

2.5 Sleepy Channels: Our Bi-Directional Payment Channel Protocol

In this section, we describe our Sleepy Channel protocol for realizing bi-directional payment channels for a currency whose transaction scheme makes use of the signature scheme Π_{DS} for authentication. For simplicity, we assume the transaction scheme lets verify transaction timeouts[4], meaning that a transaction is considered valid only if it is posted after a specified timeout $\mathbf{T}$ has passed. We discuss in Section 2.5.2 how we can remove this assumption from the transaction scheme. We additionally make use of 2-party protocols whose functionality we describe below.

2-Party key generation. Parties A and B can jointly generate keys for a signature scheme Π_{DS}. We denote this interactive protocol by Γ_{JKGen}. It takes as input the public parameters pp from both parties and outputs the joint public key pk to both parties and outputs the secret key share sk_A to A and sk_B to B.

[4]Realizable through the locktime script that is available in Bitcoin.

2-Party signing. Parties A and B having a shared key can jointly sign messages with respect to the signature scheme Π_{DS}. We denote this interactive protocol by Γ_{Sign}. It takes as input the message m and the shared public key pk from both parties and secret key shares sk_A and sk_B from A and B, respectively. The protocol outputs the signature σ (to one of the parties), such that $\Pi_{\mathsf{DS}}.\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1$.

We can instantiate both 2-party protocols (Γ_{JKGen} or Γ_{Sign}) with efficient interactive protocols for specific signature schemes of interest. If the currencies use ECDSA signatures, Schnorr signatures or BLS signatures [BLS01, Chi] for transaction authentication, we can instantiate Γ_{JKGen} and Γ_{Sign} with protocols from [Lin21], [GJKR99], or [BDN18], respectively. Monero uses a linkable ring signature scheme [TMSS22, MSBL+20] for authentication and the corresponding tailored 2-party protocols for key generation and signing are described in [TMSS22].

2.5.1 Our Protocol

We consider parties A and B already have an open channel Ch_{AB} which is a shared public key pk_{AB} (between A and B) and the corresponding secret key sk_{AB} is shared among the parties. Parties can make multiple payments using the channel (in either direction) and confirm the final payment state on the chain. However, after each payment, the payment state of the channel is updated and accordingly old states are revoked. The formal description of the protocol can be found in Figure 2.5.

High Level Overview

We present below the intuition for our protocol in prose and refer to Figure 2.3 in Section 2.2 for the transaction flow of the construction.

Payment. For each payment from the channel Ch_{AB}, parties generate two versions of transactions, $\mathsf{tx}^A_{\mathsf{Pay}}$ and $\mathsf{tx}^B_{\mathsf{Pay}}$, one version under the control of party A and the other in the control of party B. By "under control", we mean that in party A's version, A has the necessary signatures to post the payment transaction $\mathsf{tx}^A_{\mathsf{Pay}}$. Analogously, B has the necessary signature to post the payment transaction $\mathsf{tx}^B_{\mathsf{Pay}}$. Both of these transactions spend from Ch_{AB}. In contrast to prior bi-directional protocols, both versions have an important asymmetry in the coin distribution among the parties.

In more detail, the channel Ch_{AB} holds in total $f + 2c$ coins where f is the payment capacity among the parties, while $2c$ is the collateral amount locked by both parties A and B with c coins from each. The value of c is agreed upon by the parties locally before they open the channel and are returned to the respective parties at the close of the channel. Consider a payment where A's balance is v_A and B's is v_B such that $v_A + v_B = f$. The payment transaction $\mathsf{tx}^A_{\mathsf{Pay}}$ splits the funds of Ch_{AB} in the following way: (1) c coins to an address fully controlled by A, (2) v_A coins to a shared address between A and B referred to as the sleepy channel $\mathsf{SleepyCh}_A$, and (3) $v_B + c$ coins to a shared address between A and B referred to as the exit channel ExitCh_A.

Notice that A can immediately get c coins from output (1). To spend from output (2) (the sleepy channel $\mathsf{SleepyCh}_A$) which is a shared address, parties sign 2 different transactions.

1. Transaction $\mathsf{tx}^{A,A}_{\mathsf{Fpay}}$, that transfers v_A to an address of A, but is valid only after a timeout $\mathbf{T}$.

2. Transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay}}$, that spends from $\mathsf{SleepyCh}_A$ and an auxiliary address aux_A (contains ϵ coins as output in $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$, see below) that is also a shared address between A and B. The transaction transfers v_A coins from $\mathsf{SleepyCh}_A$ and ϵ (a negligible amount) from aux_A, to an address of A.

The signatures on both of the above transactions are possessed by A and not B.

To spend from output (3) (the exit channel ExitCh_A) which is a shared address, parties sign a transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$ that transfers ϵ coins to the auxiliary address aux_A and $v_B + c - \epsilon$ coins to an address of B. Notice that B's balance v_B and its collateral c (minus a negligible amount ϵ) are transferred together to B's address. In contrast to output (2), the signature on $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$ is only available with B and not A. The version for B following $\mathsf{tx}^{B}_{\mathsf{Pay}}$ is analogous to what we saw above except the roles are reversed.

Close. To close the channel with this payment state, we have two scenarios where either both parties are responsive, or one of them is unresponsive. For simplicity, we consider A as the party closing the channel and B is either responsive or not. If B is responsive, party A posts $\mathsf{tx}^{A}_{\mathsf{Pay}}$ with the corresponding signature that it has, on the blockchain. Since B is responsive, it posts the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$ spending from ExitCh_A with the corresponding signature that it has, on the blockchain. Note that B now retrieves its balance v_B and collateral c, while one of the outputs of the transaction is aux_A. Now party A can finish the payment fast, by posting the transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay}}$ that spends from $\mathsf{SleepyCh}_A$ and aux_A simultaneously, thus retrieving its balance v_A (plus some ϵ). Recall that A can already retrieve its collateral c by itself.

In the latter case where B is unresponsive, party A posts $\mathsf{tx}^{A}_{\mathsf{Pay}}$ on the blockchain as above. Now, A waits until the timeout $\mathbf{T}$ and posts the transaction $\mathsf{tx}^{A,A}_{\mathsf{Fpay}}$ that retrieves v_A coins from $\mathsf{SleepyCh}_A$ to itself. Party B can retrieve $v_B + c - \epsilon$ coins from ExitCh_A anytime it wishes.

Payment revocation and punishment. When the parties want to revoke the payment, they together generate a punishment transaction $\mathsf{tx}^{A}_{\mathsf{Pnsh}}$ that spends from $\mathsf{SleepyCh}_A$ to an address of B. The parties generate a signature on this transaction such that B holds the signature. Similar punishment transaction and signature are generated in B's version where A holds the signature for the transaction. In total, the parties have three different transactions spending from the sleepy channel $\mathsf{SleepyCh}_A$.

If party A misbehaves, and posts $\mathsf{tx}^{A}_{\mathsf{Pay}}$ after it has been revoked, party B has until timeout $\mathbf{T}$ to punish this behavior by posting $\mathsf{tx}^{A}_{\mathsf{Pnsh}}$ and the corresponding signature.

This results in B getting the v_A coins. Party B then posts the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay}}$ spending from ExitCh_A retrieving $v_B + c - \epsilon$. In effect, A only gets its collateral back, while B is able to retrieve the entire payment capacity f and its own collateral c.

Security

In this section, we state our main theorem and we informally outline the main steps our our analysis. In Appendix A.1 we give a formal description of our Sleepy Channels protocol Π in the UC framework. It differs from the protocol Π'' in Section 2.5 in that the cryptographic protocols for 2-party key generation and 2-party signing are substituted by the corresponding ideal functionalities. This is captured by the following Lemma.

Lemma 1. *Let Γ_{JKGen} be a UC-secure 2-party key-generation protocol and let Γ_{Sign} be a UC-secure 2-party signing protocol. Then the protocols Π and Π'' are computationally indistinguishable from the point of view of the environment $\mathcal{E}$.*

In Appendix A.1.1 we describe a simulator $\mathcal{S}$ that interacts with the ideal functionality $\mathcal{F}$ (defined in Section 2.4), whereas the environment interacts with $\phi_{\mathcal{F}}$ (the ideal protocol for $\mathcal{F}$). Then in Appendix A.1.2 we show that any attack that can be carried out against Π can also be carried out against $\phi_{\mathcal{F}}$. This allows us to state the following theorem.

Theorem 1. *The protocol Π UC-realizes the the ideal functionality $\mathcal{F}_L$.*

2.5.2 Discussion

In this section, we discuss key aspects of our collateral requirement and describe extensions of our protocol that make it applicable in a wider class of settings.

Collateral as incentive. Observe that the collateral of the party initiating the closing is retrieved by that party during the closing, irrespective of a cheating event. This is because the purpose of the collateral in the Sleepy Channels protocol is to incentivize fast closure of the channel by the other party if one of the parties wishes to close the channel and the other party happens to be online. Notice that if party A wishes to close the channel with an unrevoked payment, it posts the corresponding payment transaction $\mathsf{tx}^{A}_{\mathsf{Pay}}$ on the chain. Now, A immediately retrieves its collateral c, while A's channel balance v_A, and B's channel balance and collateral, i.e., $v_B + c$ are still lying unspent in the outputs of $\mathsf{tx}^{A}_{\mathsf{Pay}}$. If the value of c is high enough, party B is discouraged from launching a DoS attack on A: where party B does not retrieve the coins from ExitCh_A and lets party A wait until the timeout $\mathbf{T}$ to get v_A back. To see this, if party B attempts to launch the DoS attack on A, party B itself locks $v_B + c - \epsilon$ coins in ExitCh_A until $\mathbf{T}$. On the other hand, if B retrieves its coins from ExitCh_A immediately, party A also can retrieve its coins from $\mathsf{SleepyCh}_A$ immediately with the aid of aux_A.

The value of c is determined by the level of trust between A and B. If both parties completely trust each other, the collateral c is set to 0. In the worst case where they do

Parties A and B have a payment channel Ch_{AB} with capacity $f + 2c$ and secret key share for the channel are $\mathsf{sk}^{A}_{\mathsf{Ch},AB}$ and $\mathsf{sk}^{B}_{\mathsf{Ch},AB}$ for party A and B, respectively. Here f denotes the payment capacity of the channel and c is the collateral that a party allocates for the channel. Parties additionally have a refund transaction $\mathsf{tx}_{\mathsf{rfnd}} := \mathsf{tx}(\mathsf{Ch}_{AB}, [\mathsf{pk}_A, \mathsf{pk}_B], [v_A + c, v_B + c])$ and the corresponding signature σ_{rfnd} with respect to Ch_{AB}, where $v_A + v_B = f$ and pk_A and pk_B are some public keys of A and B, respectively.

Address Generation:

1. Parties generate the following key pairs using $\Pi_{\mathsf{DS}}.\mathsf{Gen}(1^n)$
   - Party A generates $\left(\mathsf{pk}_{\mathsf{CPay},A}, \mathsf{sk}_{\mathsf{CPay},A}\right)$, $(\mathsf{pk}_{\mathsf{pun},A}, \mathsf{sk}_{\mathsf{pun},A})$, $(\mathsf{pk}_{\mathsf{fp},A}, \mathsf{sk}_{\mathsf{fp},A})$ and $(\mathsf{pk}_{\mathsf{ffp},A}, \mathsf{sk}_{\mathsf{ffp},A})$
   - Party B generates $\left(\mathsf{pk}_{\mathsf{CPay},B}, \mathsf{sk}_{\mathsf{CPay},B}\right)$, $(\mathsf{pk}_{\mathsf{pun},B}, \mathsf{sk}_{\mathsf{pun},B})$, $(\mathsf{pk}_{\mathsf{fp},B}, \mathsf{sk}_{\mathsf{fp},B})$ and $(\mathsf{pk}_{\mathsf{ffp},B}, \mathsf{sk}_{\mathsf{ffp},B})$
2. Parties run Γ_{JKGen} to generate shared addresses: $\mathsf{SleepyCh}_A, \mathsf{SleepyCh}_B, \mathsf{ExitCh}_A, \mathsf{ExitCh}_B, \mathsf{aux}_A, \mathsf{aux}_B$.

---

i-th Payment: For the i-th payment where $v_{A,i}$ and $v_{B,i}$ are the balance of A and B, respectively with $f = v_{A,i} + v_{B,i}$, the parties do the following:

Payment Transactions: Generate payment transactions

$\mathsf{tx}^{A}_{\mathsf{Pay},i} := \mathsf{tx}\left(\mathsf{Ch}_{AB}, [\mathsf{pk}_{\mathsf{CPay},A}, \mathsf{SleepyCh}_A, \mathsf{ExitCh}_A], [c, v_{A,i}, v_{B,i} + c]\right)$ and

$\mathsf{tx}^{B}_{\mathsf{Pay},i} := \mathsf{tx}\left(\mathsf{Ch}_{AB}, [\mathsf{pk}_{\mathsf{CPay},B}, \mathsf{SleepyCh}_B, \mathsf{ExitCh}_B], [c, v_{B,i}, v_{A,i} + c]\right)$

Punishment Transactions: Generate $\mathsf{tx}^{A}_{\mathsf{Pnsh},i} := \mathsf{tx}\left(\mathsf{SleepyCh}_A, \mathsf{pk}_{\mathsf{pun},B}, v_{A,i}\right)$ and $\mathsf{tx}^{B}_{\mathsf{Pnsh},i} := \mathsf{tx}\left(\mathsf{SleepyCh}_B, \mathsf{pk}_{\mathsf{pun},A}, v_{B,i}\right)$

Finish-Payment Transactions:

1. Generate $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i} := \mathsf{tx}\left(\mathsf{SleepyCh}_A, \mathsf{pk}_{\mathsf{fp},A}, v_{A,i}\right)$ and $\mathsf{tx}^{B,B}_{\mathsf{Fpay},i} := \mathsf{tx}\left(\mathsf{SleepyCh}_B, \mathsf{pk}_{\mathsf{fp},B}, v_{B,i}\right)$ both timelocked until time $\mathbf{T}$.
2. Generate another set of faster finish-pay transactions $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i} := \mathsf{tx}\left(\mathsf{ExitCh}_A, [\mathsf{pk}_{\mathsf{ffp},B}, \mathsf{aux}_A], [v_{B,i} + c - \epsilon, \epsilon]\right)$ and $\mathsf{tx}^{B,A}_{\mathsf{Fpay},i} := \mathsf{tx}\left(\mathsf{ExitCh}_B, [\mathsf{pk}_{\mathsf{ffp},A}, \mathsf{aux}_B], [v_{A,i} + c - \epsilon, \epsilon]\right)$.
3. Generate a set of enabler transactions $\mathsf{tx}^{A*}_{\mathsf{Fpay},i} := \mathsf{tx}\left([\mathsf{SleepyCh}_A, \mathsf{aux}_A], \mathsf{pk}_{\mathsf{fp},A}, v_{A,i} + \epsilon\right)$ and $\mathsf{tx}^{B*}_{\mathsf{Fpay},i} := \mathsf{tx}\left([\mathsf{SleepyCh}_B, \mathsf{aux}_B], \mathsf{pk}_{\mathsf{fp},B}, v_{B,i} + \epsilon\right)$ that enable a faster finish-payment.

Signature Generation: Parties generate signatures on transactions by running the interactive protocol Γ_{Sign} in each step. In case one of the parties aborts at any step, the other party closes the channel with the $(i-1)$-th payment state.

1. Party A receives signature $\sigma^{A,A}_{\mathsf{Fpay},i}$ on transaction $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}$ under the shared key $\mathsf{SleepyCh}_A$. Party B receives signature $\sigma^{B,B}_{\mathsf{Fpay},i}$ on transaction $\mathsf{tx}^{B,B}_{\mathsf{Fpay},i}$ under the shared key $\mathsf{SleepyCh}_B$.
2. Party A receives signatures $\left(\sigma_{\mathsf{SleepyCh},A}, \sigma_{\mathsf{aux},A}\right)$ on the transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay},i}$ with respect to the shared keys $\mathsf{SleepyCh}_A$ and aux_A, respectively. Party B receives signatures $\left(\sigma_{\mathsf{SleepyCh},B}, \sigma_{\mathsf{aux},B}\right)$ on the transaction $\mathsf{tx}^{B*}_{\mathsf{Fpay},i}$ with respect to the shared keys $\mathsf{SleepyCh}_B$ and aux_B, respectively.
3. Party A receives signature $\sigma^{B,A}_{\mathsf{Fpay},i}$ on the transaction $\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}$ under the shared key ExitCh_B. Party B receives signature $\sigma^{A,B}_{\mathsf{Fpay},i}$ on the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ under the shared key ExitCh_A.
4. Party A receives signature $\sigma^{A}_{\mathsf{Pay},i}$ on the transaction $\mathsf{tx}^{A}_{\mathsf{Pay},i}$ under the shared key Ch_{AB}. Party B receives signature $\sigma^{B}_{\mathsf{Pay},i}$ on the transaction $\mathsf{tx}^{B}_{\mathsf{Pay},i}$ under the shared key Ch_{AB}.

Revocation: To revoke the i-th payment, parties jointly generate signatures by running the interactive protocol Γ_{Sign}: Generate signature $\sigma^{A}_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^{A}_{\mathsf{Pnsh},i}$ (party A receives $\sigma^{A}_{\mathsf{Pnsh},i}$ as output and gives it to B) and signature $\sigma^{B}_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^{B}_{\mathsf{Pnsh},i}$ (party B receives $\sigma^{B}_{\mathsf{Pnsh},i}$ as output and gives it to A). If during the revocation either party aborts, the non-aborting party immediately closes the channel with the most recent unrevoked payment.

Channel Closing: Either party can close the channel Ch_{AB} with the j-th unrevoked payment. To do this:

1. Party A posts $\left(\mathsf{tx}^{A}_{\mathsf{Pay},j}, \sigma^{A}_{\mathsf{Pay},j}\right)$ on $\mathcal{L}$. This is followed by one of the two cases:
   a) Fast finish: Party B posts $\left(\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}, \sigma^{A,B}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$, and party A posts $\left(\mathsf{tx}^{A*}_{\mathsf{Fpay},i}, \sigma^{A*}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$ for fast finish
   b) Lazy finish: If not, A can post $\left(\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}, \sigma^{A,A}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$ after timeout $\mathbf{T}$
2. Analogously, party B can post $\left(\mathsf{tx}^{B}_{\mathsf{Pay},j}, \sigma^{B}_{\mathsf{Pay},j}\right)$ on $\mathcal{L}$. This is followed by one of the two cases:
   a) Fast finish: Party A posts $\left(\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}, \sigma^{B,A}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$, and party B posts $\left(\mathsf{tx}^{B*}_{\mathsf{Fpay},i}, \sigma^{B*}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$ for fast finish
   b) Lazy finish: If not, B can post $\left(\mathsf{tx}^{B,B}_{\mathsf{Fpay},i}, \sigma^{B,B}_{\mathsf{Fpay},i}\right)$ on $\mathcal{L}$ after timeout $\mathbf{T}$

Punishing Revoked payments: If A posts the j-th revoked payment $\mathsf{tx}^{A}_{\mathsf{Pay},j}$ on $\mathcal{L}$, B can post the punishment transaction $\left(\mathsf{tx}^{A}_{\mathsf{Pnsh},i}, \sigma^{A}_{\mathsf{Pnsh},i}\right)$ on $\mathcal{L}$ before the absolute timeout $\mathbf{T}$. If B posts the j-th revoked payment $\mathsf{tx}^{B}_{\mathsf{Pay},j}$ on $\mathcal{L}$, A can post $\left(\mathsf{tx}^{B}_{\mathsf{Pnsh},i}, \sigma^{B}_{\mathsf{Pnsh},i}\right)$ on $\mathcal{L}$ before the absolute timeout $\mathbf{T}$.

Figure 2.5: Sleepy Channel protocol - Payment setup, payments, closing, and punishment

not trust each other at all, the collateral is set to be equal to the payment capacity, i.e., $c = f$ and have $v_A \leq v_B + c - \epsilon$ when $\epsilon \approx 0$. This means that during the DoS attack, party B locks at least the same amount of coins in ExitCh_A as party A does in $\mathsf{SleepyCh}_A$. Therefore, by not letting A spend its coins until timeout $\mathbf{T}$, party B also can not spend at least the same amount of coins until timeout $\mathbf{T}$.

Asymmetric collateral. Consider the case where A has significantly more money than party B (e.g., A is a merchant and B is one of A's customers). In this case, party A may be able to easily afford to lock a collateral value c (same as B) to prevent party B from getting its coins back before time $\mathbf{T}$. To account for this apparent disparity in the financial strength between parties A and B, we can instantiate our Sleepy Channels with both parties locking different amounts of collateral. In our example, party A and party B open their channel in such a manner that A locks collateral amount c_A that is higher than the collateral amount c_B locked by party B. c_A could theoretically even be larger than the full channel capacity. This strongly discourages party A (i.e., more than when using smaller or equal collateral to that of B) from denying party B a fast channel closure. We note that our Sleepy Channels protocol is flexible in how the parties set each other's collateral before opening their channel.

Punishment cost. Note that if party A misbehaves and posts a revoked payment on the chain, party B has until time $\mathbf{T}$ to punish this behavior on the chain. It is possible that the punishment transaction posted by B costs more in terms of transaction fee than what it stands to gain after the punishment if, for example, the revoked payment is a very small coin transfer. To account for that, A (i.e., the party creating and funding the channel) can unconditionally include a certain amount for B to cover such transaction fees, as it is currently implemented in the Lightning Network [dev]. We emphasize that this is an issue that is present throughout off-chain solutions [DW15, DRO, Liga, AEE+21] including ZK-rollups [ZK-].

TimeLock script independence. The curious reader may wonder whether our protocol achieves the sought-after goal of (bi-directional) payment channels needing *only* the signature verification script from the underlying blockchain. Although we remove the dependency on *relative* timelock scripts, our protocol still relies on *absolute* timelock scripts (see point 1 in finish-payment transactions Figure 2.5) to guarantee the closure of the channel after some (fixed) time $\mathbf{T}$. Thus a natural question is whether one can construct bidirectional payment channels without relying on time-lock scripts *at all*. It turns out that, if one is willing to rely on time-lock puzzles [RSW96], we can avoid the dependence on timelock scripts entirely. As it was shown in prior works [TBM+20, TMSS22], absolute time-locks[5] can be simulated using verifiable timed signatures (VTS): VTS allow one to encapsulate a signature on a message for a pre-determined amount of time $\mathbf{T}$. At the same time, the party who is solving the puzzle, is guaranteed that the signature recovered after

[5] Crucially, this transformation does not work for the relative time-lock logic, since there the time depends on some event which is triggered by the attacker and thus one cannot set the time parameter of the VTS ahead of time.

time $\mathbf{T}$ is a valid one. Parties are required to perform persistent background computation for the lifetime of the channel. However, for currencies like Monero where we do not have any timelock script, we do not know of any other viable mechanism other than the one using VTS from [TMSS22]. A recent work [TGB+21] has enabled parties to securely outsource this computation to a decentralized network thereby removing any sort of computational load on the parties.

Extending lifetime and capacity of the channel. In contrast to Lightning Network channels, the channel Ch_{AB} between A and B is time-bounded because of the bound required in Sleepy Channels. More precisely, parties have to close the channel Ch_{AB} before the timeout $\mathbf{T}$ that are set on the finish-payment transactions $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}$ and $\mathsf{tx}^{B,B}_{\mathsf{Fpay},i}$ that spend from $\mathsf{SleepyCh}_A$ and $\mathsf{SleepyCh}_B$, respectively. However, if both parties cooperate, they can easily extend their channel duration by transferring the coins from the current channel Ch_{AB} to a new channel Ch'_{AB} (shared between A and B) in accordance with the latest channel balance that the parties had in Ch_{AB}. In other words, parties can post a single transaction on the blockchain anytime before $\mathbf{T}$ to transfer the coins from Ch_{AB} to Ch'_{AB}. The channel balance of the parties in Ch'_{AB} is set according to the most recent payment state between them in the channel Ch_{AB}. A similar procedure is adopted in the Splicing protocol [Rus18] of Lightning Network where users can periodically increase or decrease their channel capacity on-chain without violating any payments already made. Our Sleepy Channel protocol apart from extending the channel lifetime, can also update the channel capacity with this approach.

2.6 Performance Evaluation

We evaluated a proof of concept to show (i) correctness of our scheme, (ii) compatibility with Bitcoin, and (iii) on- and off-chain transaction overhead. The source code is available at [Gita].

Implementation subtleties. There are several approaches on how Sleepy Channels can be implemented, given the scripting functionality of, say, Bitcoin. For instance, timelocks can be enforced either at a single transaction output or for the whole transaction, 2-party signing can be replaced with a multisig script (for a blow-up in the transaction size) and revocation can be done via exchanging a hash secret, a private key or a signed punishment transaction upon revoking an old state. In this section, we follow our protocol as in Figure 2.5 and use transaction level timelocks, 2-party signing and exchange signed punishment transactions for revocation.

Deploying the transactions. Now we describe the transactions used in Sleepy Channels and we refer the reader to Table A.1 in Appendix A.2 for the details on transaction sizes and their cost in terms of on-chain fees. We also give a pointer to the corresponding transactions deployed in the Bitcoin testnet, thereby demonstrating the backwards compatibility of Sleepy Channels.

The first step in Sleepy Channels is building a funding transaction tx_F [Fun]. Built on top of the funding, we look at A's commitment (or state) transaction $\mathsf{tx}^{A}_{\mathsf{Pay},i}$ [Payb] and note that the transactions for B are symmetric. When A puts the current state on the ledger, there are two ways how A can claim its money. On the one hand, if B unlocks its own funds by putting $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ [Payc], then A can claim its funds with $\mathsf{tx}^{A*}_{\mathsf{Fpay},i}$ right away [Paya]. On the other hand, after the lifetime expires, A can unilaterally claim its funds with $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}$. If A puts an old state, then B can punish A via $\mathsf{tx}^{A}_{\mathsf{Pnsh},i}$. Finally, two users can close their channel honestly with a transaction, where both funds are unlocked right away.

We find that for opening a channel in Sleepy Channels, the two parties together need to put 338 bytes on-chain and exchange 2026 bytes (8 transactions off-chain). For each subsequent update, the two parties need to exchange 2408 bytes (10 transactions off-chain). The closing and punishment happen on-chain. For the closing, there are three options. Either they close honestly (225 bytes, 1 tx), or one party closes unilaterally and unlocks its funds after the timelock expires (449 bytes, 2 tx), or one party closes unilaterally and the other one unlocks the funds right away (823 bytes, 3 tx). The punishment case requires 450 bytes and 2 transactions.

Comparison to LN. As for our construction, the LN channel functionality can be implemented with subtle differences, resulting in different outcomes. The funding transaction of LN is identical to ours, except that it locks no additional collateral. The commitment transactions differ, as they have one fewer output, and therefore only 226 bytes. Moreover, in LN there are no fast finish transactions. This totals to 338 bytes on-chain and exchanging 832 bytes (4 transactions) for opening a LN channel. For updating, the users exchange 1214 bytes (6 transactions). Note that the honest, the unilateral close, and the punishment in sleepy channels are identical to LN, both in terms of transaction structure and in size.

Overhead. The Sleepy Channels protocol does not require costly cryptography. It requires computing and verifying signatures locally, 2-party signing, and a maximum off-chain communication in the order of 10^3 bytes for each operation. The computational time can be expected to be negligible on even commodity hardware; the communication is limited only by network latency.

2.6.1 Simulation

We perform some additional experiments with respect to a recent snapshot of LN (January 2022). In this snapshot, there are 81k channels, 19k channel nodes, and a total capacity of 2990 BTC. As the balance distribution of each channel is unknown, we assume that it is split evenly between the two users. The source code of our simulation experiments including the snapshot is available at [Gitb]. We repeat the experiments 100 times for each and plot the average and standard deviation.

Watchtower collateral. We investigate the collateral a watchtower service needs to provide, in order to cover their customers should they go offline. We analyze watchtower

constructions that fully collateralize the channels, e.g., [ATLW20, MSYS21, MBB+19a]. For this, we randomly sample a percentage of nodes that wish to employ a watchtower, and based on their balances in their channels, we plot the amount of collateral in Figure 2.6. This amount rises linearly with the amount of users that wish to employ a watchtower. If 30% of all users do so, (i) the watchtower service needs to lock up approximately 890 BTC and (ii) users need to pay fees for that, even if there are no disputes. Currently, this total capacity that has to be available to the watchtower service as collateral amounts to roughly 39M USD.

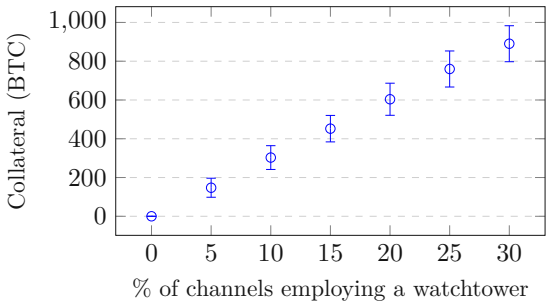

Figure 2.6: Results of the first simulation.

Risk of failing to go online. We simulate the risk of users having to periodically monitor the blockchain in LN. In LN, the time frame for punishment is one day (144 blocks). I.e., in this time users need to come online at least once and check whether or not the other party tried to cheat. In our setting, we investigate a time period of 30 days with users trying to come online each day.

In our simulation, we assume that there is a certain chance that users fail to come online and monitor the blockchain in a given time frame, e.g., due to power outages, DoS attacks, etc. We further assume that neighboring nodes will notice this; a realistic assumption due to the *ping and pong* messages [Ligb] of the LN. We assume that neighboring nodes want to maximize their profits and will exploit such a case by putting an old state and thereby, potentially stealing funds of the offline user.

The Sleepy Channels protocol would not fully prevent this behavior, but reduce it significantly. That is, for a given period of time, in this simulation 30 days, the users need to come online only once, e.g., before the channel expires. They can of course fail to come online there with the same probability, but this event occurs only once instead of 30 times. Obviously, the longer this time span is, the greater the chances for LN nodes is to miss at least one of these intervals, while for Sleepy Channels it remains the same. For 30 days, only about 3% of the channels are at risk for Sleepy Channels compared to LN, for any given chance of missing the online check.

In Figure 2.7 we plot the number of channels that are at risk for a given chance that a user will fail to come online in each interval, once for each the LN and Sleepy Channels. The y axis is shown in a logarithmic scale. Over a one-month period, there are 5k channels

(0.1% chance) and 49k (1% chance) channels at risk (roughly 60% of the LN) for LN channels. For Sleepy Channels, these numbers are 170 channels (0.1% chance) and 1.9k channels (1% chance).

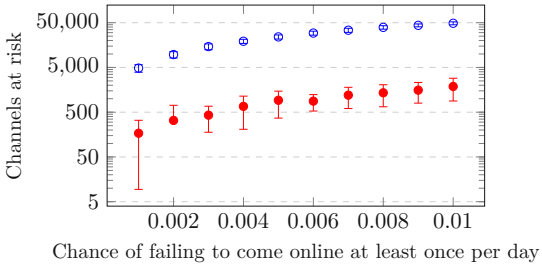

Figure 2.7: Results of the second simulation. (Blue = LN, Red = Sleepy Channels)

2.7 Conclusion

Payment channels are one of the most promising payment solutions for blockchain-based cryptocurrencies. Despite their large adoption, many such proposals suffer from limitations, such as requiring the parties to be constantly online and monitor the network, or outsourcing this task to third parties (e.g., watchtowers). In this work, we propose a new payment channel architecture (Sleepy Channels) that supports bi-directional payments and does not require the parties to be persistently online. The protocol is backward compatible with many existing currencies (e.g., Bitcoin, Monero...) and relies on lightweight cryptographic machinery. Our performance evaluation shows that the protocol is efficient enough to be adopted in large payment ecosystems (such as the Lightning Network). An interesting open question is whether our techniques are also applicable to account-based currencies, rather than UTXO-based currencies.

Acknowledgments

This work has been also partially supported by Madrid regional government as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union, by grant IJC2020-043391-I/MCIN/AEI/10.13039/501100011033 and European Union NextGenerationEU/PRTR, by SCUM Project (RTI2018-102043-B-I00) MCIN/AEI/10.13039/501100011033/ERDF A way of making Europe, and by the project HACRYPT. This work was also partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the projects PROFET (grant agreement P31621), by the Austrian Research Promotion Agency (FFG) through the COMET K1 SBA and COMET K1 ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things

(CDL-BOT) and by CoBloX Labs. This work was further partially supported by the German Federal Ministry of Education and Research BMBF (grant 16K15K042, project 6GEM). The work was also funded through the support of THE DAVID AND LUCILLE PACKARD FOUNDATION - Award #202071730, SRI INTERNATIONAL - Award #53978 / Prime: DEFENSE ADVANCED RESEARCH PROJECTS AGENCY - Award #HR00110C0086, NATIONAL SCIENCE FOUNDATION - Award #2212746.

CHAPTER 3

Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures

Abstract

Decentralized and permissionless ledgers offer an inherently low transaction rate, as a result of their consensus protocol demanding the storage of each transaction on-chain. A prominent proposal to tackle this scalability issue is to utilize off-chain protocols, where parties only need to post a limited number of transactions on-chain. Existing solutions can roughly be categorized into: (i) application-specific channels (e.g., payment channels), offering strictly weaker functionality than the underlying blockchain; and (ii) state channels, supporting arbitrary smart contracts at the cost of being compatible only with the few blockchains having Turing-complete scripting languages (e.g., Ethereum).

In this work, we introduce and formalize the notion of *generalized channels* allowing users to perform any operation supported by the underlying blockchain in an off-chain manner. Generalized channels thus extend the functionality of payment channels and relax the definition of state channels. We present a concrete construction compatible with any blockchain supporting transaction authorization, time-locks, and constant number of Boolean $\wedge$ and $\vee$ operations – requirements fulfilled by many (non-Turing-complete) blockchains including the popular Bitcoin. To this end, we leverage *adaptor signatures* – a cryptographic primitive already used in the cryptocurrency literature but formalized as a standalone primitive in this work for the first time. We formally prove the security of our generalized channel construction in the Universal Composability framework.

As an important practical contribution, our generalized channel construction outperforms the state-of-the-art payment channel construction, the Lightning Network, in efficiency.

Concretely, it halves the off-chain communication complexity and reduces the on-chain footprint in case of disputes from linear to constant in the number of off-chain applications funded by the channel. Finally, we evaluate the practicality of our construction via a prototype implementation and discuss various applications including financially secured fair two-party computation.

This chapter presents the results of a collaboration with Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi, which was published at the 27th Annual International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt) in 2021 under the title "Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures". I am responsible for writing a proof-of-concept implementation, deploying and testing the scheme on the Bitcoin testnet, designing and conducting the experiments and measurements as well as comparison to the Lightning Network. Further, I contributed to the write-up of the paper. Kristina Hostáková proposed the concept of generalized channels and is responsible for the ideal functionality. Oğuzhan Ersoy and Kristina Hostáková are responsible for the application section. Siavash Riahi is responsible for formalizing adaptor signatures. Siavash Riahi and Andreas Erwig are responsible for the security proofs. Pedro Moreno-Sanchez, Sabastian Faust, and Matteo Maffei were the general advisors and contributed with continuous feedback.

3.1 Introduction

One of the most fundamental technical challenges of decentralized and permissionless blockchains is scalability. Since transactions are processed via a costly distributed consensus protocol run among a set of parties (so-called miners), transaction throughput is limited and transaction confirmation is slow. There has been a plethora of work on improving the scalability of blockchains, with off-chain protocols being one of the most promising solutions.

Intuitively, off-chain protocols build a second layer over the blockchain (often referred to as the *layer-1*) by allowing the vast majority of transactions to be processed directly between the involved participants, with the blockchain being used only in the initial setup and in case of disputes, thereby drastically improving transaction throughput and confirmation time.

While there exists a large variety of different off-chain (or layer-2) solutions (see, e.g., [BSA+17, WSNH19, GMSR+20, JKLT19] and many more), *payment channels* [Bit18, DW15, PD16] are by far the most prominent one. Intuitively, a payment channel works in three phases. First, the two users *open* a channel by locking a certain amount of coins on-chain into an account controlled by both users. Then they perform an arbitrary amount of payments by exchanging authenticated messages *off-chain*. Finally, they *close* the channel by announcing the outcome of their trades to the ledger.

Off-chain computations in ethereum. Ethereum supports on-chain transactions specified in a *Turing-complete scripting language*, which enables the execution of arbitrarily

complex programs, also called smart contracts, thereby going beyond simple payments. The underlying blockchain is organized accordingly in the account-based model, in which the balance associated with an account is explicitly stored in its memory and programmatically updated via smart contracts. By leveraging the expressiveness of Turing-complete scripting languages, payment channels can be generalized into so-called *state channels* [MBB+19b, DFH18, DEF+19b], whose functionality goes far beyond simple payments. Namely, state channels enable users to execute arbitrarily complex smart contracts in an off-chain manner, thereby making their execution faster and cheaper.

Turing-complete vs restricted scripting. The majority of current blockchains (e.g., Bitcoin, Zcash, Monero, and Cardano's ADA) only support a restricted scripting language and are based on the Unspent Transaction Output (UTXO) model: intuitively, they enable a restricted class of transactions, possibly conditioned to some events, that transfer money from an unspent transaction to a new unspent transaction. There are several reasons behind the choice of a limited scripting language. First, the simplicity of design and usage, which is believed to be beneficial for security: countless examples of smart contract vulnerabilities on Ethereum show that complex contract logic and increased expressiveness pave the way for critical bugs, which may have severe consequences for the stability of the underlying currency as shown by the infamous DAO hack [Sie16]. Second, blockchains with simple transaction logic are less costly to maintain: this is important as transaction execution is done by many parties and even normal users. Finally, restricted scripting languages are expressive enough to encode many interesting computations (e.g., lotteries [ADMM16], auctions [DDM+18], and more [BK14, KB14, BZ18]).

Unfortunately, current state channel constructions are not applicable without a Turing-complete scripting language, thereby excluding the majority of blockchains. In this work, we investigate the following question: *Can we generically lift any transaction logic offered by layer-1 to layer-2 even for blockchains with restricted transaction logic?* Besides its practical importance, we believe that this question is theoretically interesting. It may constitute a first step towards a more general research agenda exploring the feasibility (or impossibility) of generic off-chain computation from blockchains with limited expressiveness.

3.1.1 Our contribution

Our main contribution is to put forward the notion of *generalized channels* – a generic extension of payment channels to support off-chain execution of *arbitrary transaction logic* supported by the underlying blockchain. State channels can hence be seen as a special case of generalized channels for blockchains with Turing-complete scripting languages. We briefly outline our main contributions below. A technical overview of our construction is given in Section 3.2.

Generalized Channels. We show that if the underlying UTXO-based blockchain supports transaction authorization, time-locks, and basic Boolean logic (constant number

of $\wedge$, $\vee$ operations), then *any* transaction logic available on layer-1 can be lifted to layer-2 securely and generically.

As most cryptocurrencies, including the by far most prominent Bitcoin, satisfy the assumptions of our construction, they can benefit from generalized channels as a scalability solution. This, in particular, implies that our construction directly enables us to execute *any* Bitcoin transaction off-chain. Moreover, we stress that our construction can also be deployed over any blockchain that can simulate a UTXO-based system, which, in particular, includes blockchains with support for Turing-complete smart contracts, e.g., Ethereum or Hyperledger Fabric [ABB+18].

A novel revocation mechanism for Generalized Channels. The main technical challenge in our generalized channel design is to propose an efficient mechanism for old channel state revocation while putting minimal assumptions on the scripting language of the underlying blockchain. The state-of-the-art approach, put forward by the Lightning Network [PD16], uses a punishment mechanism that allows the cheated party to claim all coins from the channel. As we argue, a straightforward generalization of the Ligthning-style revocation is unsuitable for generalized channels. Firstly, the blockchain communication complexity in case of misbehavior depends on the number of parallel conditional payments funded by the channel. This significantly increases the blockchain overhead when processing a punishment (if triggered). Secondly, the security of the revocation mechanism relies on state duplication, hence each off-chain transaction funded by the channel has to be performed twice (once on each duplicate). This is particularly problematic when channels are built on top of channels [EMSM19] as the off-chain communication complexity grows exponentially with the number of channel layers.

To overcome these drawbacks, we design a novel revocation mechanism reducing the on-chain complexity in case of a dispute from linear to constant, and the off-chain communication complexity from exponential to linear.

Formalization of adaptor signatures. A key idea of our novel revocation mechanism is to utilize an *adaptor signature scheme* [Poe17] – a cryptographic primitive introduced by the cryptocurrency community to tie together the authorization of a transaction and the leakage of a secret value. Although adaptor signatures have been used in previous works (e.g. [MMS+19, Fou19, MSK]), a formal definition has never been presented. We fill this gap by providing the first formalization of adaptor signatures and their security (in terms of cryptographic games), and proving that ECDSA and Schnorr-based schemes satisfy our notions. We believe that our formalization and security analysis of adaptor signatures is of independent interest (see details on the impact of our work below).

Formalization of Generalized Channels. In order to formally define the security guarantees of a generalized channel protocol, we utilize the extended Universal Composability model allowing for global setup (the GUC model for short) put forward by Canetti et al. [CDPW07]. More precisely, we model the money mechanics of a UTXO-based blockchain via a global ledger ideal functionality and provide an ideal specification of a

generalized channel protocol via a novel ideal functionality. Thereafter, we prove that our generalized channel construction satisfies this ideal specification. The key challenges of our security analysis are to ensure the consistency of timings imposed by the blockchain processing delay and to ensure that no honest party can ever lose coins by participating in a channel.

Evaluation and applications. We implemented our protocols and conducted an experimental evaluation, demonstrating how to use generalized channels as a building block for popular off-chain applications like payment routing through a payment channel network (PCN) [PD16,MMSK$^+$17,MMS$^+$19] and channel splitting [EMSM19]. Concretely, our evaluation demonstrates that, already when routing *one* payment through a channel, the amount of blockchain fees in case of a dispute is reduced by 28% compared to the state-of-the-art Lightning network solution. In practice, there have been cases of disputes in channels with 50 concurrent payments [lnc20], which currently costs 553.66 USD in fees to resolve in Lightning and only 17.47 USD with generalized channels. For channel splitting, we reduce the transactions to be exchanged off-chain per sub-channel from exponential to constant.

Moreover, we discuss how to use generalized channels to realize the Claim-or-Refund functionality of Bentov and Kumaresan [BK14]. This functionality, can be used to build a fair two-party computation protocol over Bitcoin, where fairness is achieved by financially penalizing malicious parties. Realizing the Claim-or-Refund functionality, in particular, implies that generalized channels allow parties to execute any two-party computation off-chain.

Impact of our work. Our work has resulted in several interesting follow-up works. In the case of adaptor signatures, Esgin et al. [EEE20] and Tairi et al. [TMSM21b] have proposed adaptor signature constructions secure against adversaries with quantum computing power which allows for payment channels or atomic swaps in post-quantum secure blockchains. Recently, Erwig et al. [EFH$^+$21] showed how to generically build single and 2-party adaptor signatures from identification schemes. All these works follow our definition of adaptor signatures that we put forth in this work. Our generalized channels have also been used as a basis for virtual channel constructions in [AME$^+$21] and have recently been extended to support fair and privacy-preserving watchtowers by Mirzaei et al. [MSYS21]. We will talk in more details about some of these follow-up works in Section 3.7.

3.1.2 Other related work

We briefly discuss related work on off-chain protocols and adaptor signatures, where the latter is an important building block in our construction.

Off-chain protocols. As already mentioned before, there has been an extensive line of work on various types of payment channels [Bit18, DW15, PD16] and payment channel networks (PCNs) [PD16, MMSK$^+$17, MMS$^+$19]. However, these constructions

only support simple payments and do not extend to support more complex transaction logic. The authors in [KL19] provide a formalization of the Lightning Network (LN) in the UC framework. This formalization is, however, tailored to the details of the current LN and cannot be leveraged to formalize generalized channels as we propose here. Most related to our work is the research on state channels [MBB+19b, DFH18, DEF+19b], as these constructions allow to lift any transaction logic supported by the underlying blockchain off-chain. However, state channels crucially rely on the underlying blockchain to support smart contracts and hence do not work for blockchains with restricted scripting language. Finally, eltoo [DRO] is a payment channel construction that does not rely on a punishment mechanism, yet requires Bitcoin to adapt a new scripting command (op-code). This op-code, however, has not been included in Bitcoin's scripting language in the past due to security concerns. In the case of address reuse or lazy wallet designs, funds can be stolen by replaying transactions [Tra]. Moreover, the security of the eltoo protocol has not been formally proven and it only supports simple payments.

Apart from payment and state channels, numerous other solutions have been proposed in order to perform heavy on-chain computation off-chain. For instance, various previous works (e.g., [DEF+19a, CZK+19, KMS+16]) focus on realizing on-chain functionality off-chain by using Trusted Execution Environments which, however, inherently add additional trust assumptions that may not hold in practice (e.g., [BMD+17, CCX+19, BMW+18]). A proposal to remove these assumptions is to use MPC protocols [BK14, KB14], which however require collateral linear in the number of conditional payments. In contrast, generalized channels only require constant collateral for the execution of an arbitrary number of such payments. There have been proposals to remedy the collateral requirement in MPC protocols [BKM17, KB16, KMB15] but they are incompatible with many existing UTXO blockchains, including Bitcoin.[1]

Adaptor Signatures. Poelstra [Poe17] introduced the notion of adaptor signatures (AS), which intuitively allows to create partial signatures whose completion is conditioned on solving a cryptographic hard problem – a feature that has been proven useful in off-chain applications such as PCNs [MMS+19] and payment-channel hubs [TMSM21a]. For instance, Malavolta et al. [MMS+19] use AS as a building block to define and realize multi-hop payments in PCNs. Moreover, AS have been used as an off-the-shelf cryptographic building block for multi-path payments [EFHR20] and Monero-compatible PCNs [TMSS22]. Banasik et al. [BDM16] construct a scheme satisfying a similar notion to AS in order to allow two parties to exchange a digital asset using cryptocurrencies that do not support Turing-complete programs. None of these works, however, define AS as a stand-alone primitive. Concurrently to our work, Fournier [Fou19] attempts to formalize AS as an instance of one-time verifiable encrypted signatures [BGLS03]. Yet, the definition of [Fou19] is weaker than the one we give in this work and does not suffice for the channel applications. Also concurrent to this work, Thyagarajan and Malavolta [TM21] define *lockable signatures*. While similar to AS in spirit, lockable

[1] These solutions require the underlying blockchain to either support verification of signatures on arbitrary messages or Turing-complete smart contracts.

signatures are a weaker primitive as the partial signature must be created honestly (e.g., through MPC) and the solution to the cryptographic hardness problem must be known beforehand. On the other hand, lockable signatures can be built from any signature scheme while AS cannot be constructed from unique signatures [EFH+21].

3.2 Background and Solution Overview

Blockchain transactions. We focus on blockchains based on the Unspent Transaction Output (UTXO) model, such as Bitcoin. In the UTXO model, coins are held in *outputs*. Formally, an output θ is a tuple (cash, φ), where cash denotes the amount of coins associated with the output and φ defines the conditions (also known as scripts) that need to be satisfied to spend the output.

A *transaction* transfers coins across outputs meaning that it maps (possibly multiple) existing outputs to a list of new outputs. The existing outputs that fund the transactions are called *transaction inputs*. In other words, transaction inputs are those tied with previously unspent outputs of older transactions. Formally, a transaction tx is a tuple of the form $(\mathsf{txid}, \mathsf{input}, \mathsf{output}, \mathsf{Witness})$, where $\mathsf{txid} \in \{0,1\}^*$ is the unique identifier of tx and is calculated as $\mathsf{txid} := \mathcal{H}([\mathsf{tx}])$, where $\mathcal{H}$ is a hash function modeled as a random oracle and $[\mathsf{tx}]$ is the *body of the transaction* defined as $[\mathsf{tx}] := (\mathsf{input}, \mathsf{output})$; input is a vector of strings identifying all transaction inputs; $\mathsf{output} = (\theta_1, \ldots, \theta_n)$ is a vector of new outputs; and $\mathsf{Witness} \in \{0,1\}^*$ contains the witness allowing to spend the transaction inputs.

To ease the readability, we illustrate the transaction flows using charts (see Figure 3.1 for examples). We depict transactions as rectangles with rounded corners. Doubled-edge rectangles represent transactions published on the blockchain, while single-edge rectangles are transactions that could be published on the blockchain, but they are not (yet). Transaction outputs are depicted as a box inside the transaction. The value of the output is written inside the output box and the output condition is written above the arrow coming from the output.

Conditions of transaction outputs might be fairly complex and hence it would be cumbersome to spell them out above the arrows. Instead, for frequently used conditions, we define the following abbreviated notation. If the output script contains (among other conditions) signature verification w.r.t. some public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ on the body of the spending transaction, we write all the public keys *below* the arrow and the remaining conditions *above* the arrow. Hence, information below the arrow denotes "who *owns* the output" and information above denotes "additional spending conditions". If the output script contains a check of whether a given witness hashes to a predefined h, we express this by writing the hash value h *above* the arrow. Moreover, if the output script contains a relative time-lock, i.e., a condition that is satisfied if and only if at least t rounds passed since the transaction was published, we write "$+t$" *above* the arrow. Finally, if the output script φ can be parsed as $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ for some $n \in \mathbb{N}$, we add a diamond shape

Figure 3.1: (Left) tx is published on the blockchain. The output of value x_1 can be spent by a transaction containing a preimage of h and signed w.r.t. pk_A. The output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t rounds passed since tx was accepted by the blockchain. (Right) tx' is not published yet. Its only output can be spent by a transaction whose witness satisfies $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

to the corresponding transaction output. Each of the sub-conditions φ_i is then written above a separate arrow.

Payment channels. A payment channel [PD16] enables several payments between two users without submitting every single transaction to the blockchain. The cornerstone of payment channels is depositing coins into an output controlled by two users, who then authorize new deposit balances in a peer-to-peer fashion while having the guarantee that all coins are refunded at a mutually agreed time.

First, assume that Alice and Bob want to create a payment channel with an initial deposit of x_A and x_B coins respectively. For that, Alice and Bob agree on a *funding transaction* (that we denote by tx^{f}) that sets as inputs two outputs controlled by Alice and Bob holding x_A and x_B coins respectively and transfers them to an output controlled by both Alice and Bob (i.e., its spending condition mandates both Alice's and Bob's signature). When tx^{f} is added to the blockchain, the payment channel between Alice and Bob is effectively *open*.

Assume now that Alice wants to pay $\alpha \leq x_A$ coins to Bob. For that, they create a new *commit transaction* $\mathtt{TX}_{\mathsf{c}}$ representing the commitment from both users to the new channel state. The commit transaction spends the output of tx^{f} into two new outputs: (i) one holding $x_A - \alpha$ coins owned by Alice; and (ii) the other holding $x_B + \alpha$ coins owned by Bob. Finally, parties exchange the signatures on the commit transaction, thereby completing the channel *update*. Alice (resp. Bob) could now add $\mathtt{TX}_{\mathsf{c}}$ to the blockchain. Instead, they keep it locally in their memory and overwrite it when they agree on another commit transaction, let us denote it $\overline{\mathtt{TX}}_{\mathsf{c}}$, representing a newer channel state. This, however, leads to several commit transactions that can possibly be added to the blockchain. Since all of them are spending the same output, only one can be accepted. As it is impossible to prevent a malicious user from publishing an old commit transaction, payment channels require a mechanism punishing such behavior.

Lightning Network [PD16], the state-of-the-art payment channel for Bitcoin, implements such a mechanism by introducing *two* commit transactions, denoted $\mathtt{TX}_{\mathsf{c}}^{\mathsf{A}}$ and $\mathtt{TX}_{\mathsf{c}}^{\mathsf{B}}$, per channel update, each of which contains a punishment mechanism for one of the users. In more detail (see also Figure 3.2), the output of $\mathtt{TX}_{\mathsf{c}}^{\mathsf{A}}$ representing Alice's balance in the channel has a special condition. Namely, it can be spent by Bob if he presents a

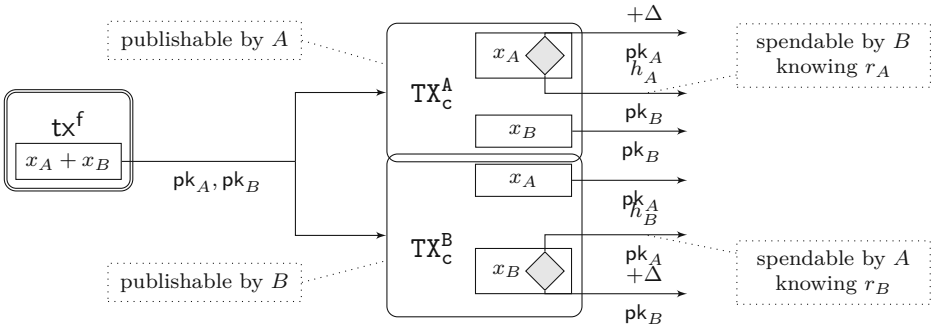

Figure 3.2: A Lightning style payment channel where A has x_A coins and B has x_B coins. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ upper bounds the time needed to publish a transaction on a blockchain.

preimage of a hash value h_A or by Alice if a certain number of rounds passed since the transaction was published. During a channel update, Alice chooses a value r_A, called the *revocation secret*, and presents the hash $h_A := \mathcal{H}(r_A)$ to Bob. Knowing h_A, Bob can create and sign the commit transaction $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$ with the built-in punishment for Alice (analogously for Bob and $\mathtt{TX}_{\mathsf{c}}^{\mathtt{B}}$). During the next channel update, parties first commit to the new state by creating and signing $\overline{\mathtt{TX}}_{\mathsf{c}}^{\mathtt{A}}$ and $\overline{\mathtt{TX}}_{\mathsf{c}}^{\mathtt{B}}$, and then *revoke* the old state by sending the revocation secrets to each other thereby enabling the punishment mechanism. If a malicious Alice now publishes the old commit transaction $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$, Bob can spend both of its outputs and claim all coins locked in the channel.

3.2.1 Solution Overview

The goal of our work is to extend the idea of payment channels such that parties can agree on *any* conditional payment that they could do on-chain and not only direct payments. Technically, this means that we want the commit transaction to contain arbitrarily many outputs with arbitrary conditions (as long as they are supported by the underlying blockchain). The main question we need to answer when designing such channels, which we call *generalized channels*, is how to implement the revocation mechanism.

Revocation per update. The first idea would be to extend the revocation mechanism explained above such that *each output* of $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$ contains a punishment mechanism for Alice (analogously for Bob). While this solution works, it has several disadvantages. If one party, say, Alice, cheats and publishes an old commit transaction $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$, Bob has to spend all outputs of $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$ to punish Alice. Although Bob could group some of them within a single transaction (up to the transaction size limit), he might be forced to publish multiple transactions thereby paying high transaction fees. Moreover, such a revocation mechanism requires a high on-chain footprint not only for $\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}$, but also for Bob getting coins from the outputs.

Our goal is to design a punishment mechanism whose on-chain footprint and potential

transaction fees are *independent of the channel state*, i.e., the number and type of outputs in the channel. To this end, we propose the *punish-then-split* mechanism which separates the punishment mechanism from the actual outputs. In a nutshell, the commit transaction $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ has now only one output dedicated to the punishment mechanism which can be spent (i) immediately by Bob, if he proves that the commit transaction was old (i.e., he knows the revocation secret r_A of Alice); or (ii) after a certain number of rounds by a *split transaction* $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ owned by both parties and containing all the outputs of the channel (i.e. representing the channel state). Hence, if $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ is published on the blockchain, Bob has some time to punish Alice if the commit transaction was old. If Bob does not use this option, any of the parties can publish the split transaction $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ representing the channel state. Analogously for $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$.

One commit transaction per channel update. Another drawback of the Lightning-style revocation mechanism is the need for two commit transactions for the same channel state. While this is not an issue for simple payment channels, for generalized channels it might cause undesirable redundancy in terms of communication and computational costs. This comes from the fact that generalized channels support arbitrary output conditions and hence can be used as a source of funding for other off-chain applications, e.g., a fair two-party computation or another off-chain channel as we discuss later in this work (see Section 3.7). Such off-chain applications would, however, have to "exist" twice. Once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ being eventually published on-chain and once considering $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$. Especially when considering channels built on top of channels, the overhead grows exponentially. Our goal is to construct generalized channels that require only one commit transaction and hence avoid any redundancy.

A naive approach to design such a single commit transaction $\mathtt{TX}_{\mathtt{c}}$ would be to "merge" the transactions $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ and $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$. Such $\mathtt{TX}_{\mathtt{c}}$ could be spent (i) by Alice if she knows Bob's revocation secret; (ii) by Bob if he knows Alice's revocation secret or (iii) by the split transaction $\mathtt{TX}_{\mathtt{s}}$ representing the channels state after some time. Unfortunately, this simple proposal allows parties to misuse the punishment mechanism as follows. A malicious Alice could publish an old commit transaction $\mathtt{TX}_{\mathtt{c}}$ and since she knows Bob's revocation secret, she could immediately try to punish Bob. To prevent such undue punishment of honest Bob, we need to make sure that Alice can use the punishment mechanism only if Bob published $\mathtt{TX}_{\mathtt{c}}$.

The main idea of how to implement this additional requirement is to force the party publishing $\mathtt{TX}_{\mathtt{c}}$ to reveal some secret, which we call *publishing secret*, that the other party could use as proof. We achieve this by leveraging the concept of an *adaptor signature scheme* – a signature scheme that allows a party to *pre-sign* a message w.r.t. some statement Y of a hard relation (at a high level, a statement/witness relation is hard, if given a statement Y is it computationally hard to find a witness y). Such pre-signature can be adapted into a valid signature by anyone knowing a witness for the statement Y. Also, it is possible to extract a witness y for Y by knowing both the pre-signature and the adapted full signature. In our context, adaptor signatures allow users of a generalized channel to express the following: "I give you my *pre-signature* on $\mathtt{TX}_{\mathtt{c}}$ that you can turn

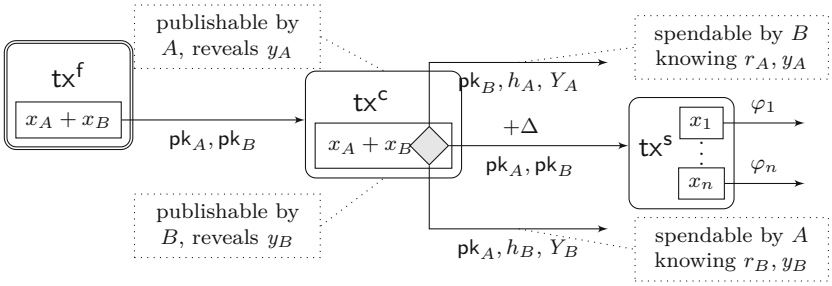

Figure 3.3: A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. In the figure, pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain.

into a full signature and publish $\mathtt{TX}_{\mathsf{c}}$, which will reveal your publishing secret to me."

To conclude, our solution, depicted in Figure 3.3, requires only one commit transaction $\mathtt{TX}_{\mathsf{c}}$ per update. The commit transaction has one output that can be spent (i) by Alice if she knows Bob's revocation secret r_B *and* publishing secret y_B; (ii) by Bob if he knows Alice's revocation secret r_A *and* publishing secret y_A or (iii) by the split transaction $\mathtt{TX}_{\mathsf{s}}$ representing the channels state after some time. In the depicted construction, we assume that statement/witness pairs used for the adaptor signature scheme are public/secret keys of the blockchain signature scheme. Hence, testing if a party knows a publishing secret can be done by requiring a valid signature w.r.t. this public key. Let us emphasize that public/secret keys can also be used for the revocation mechanism instead of the hash/preimage pairs. This is actually preferable (not only in our construction but also in the Lightning-style channels) since the punishment output script will only consist of signature verification, thereby requiring less complex scripting language. As a result, our solution does not only work over Bitcoin, but over any UTXO-based blockchain that supports transaction authorization (if there exists an adaptor signature scheme w.r.t. the considered digital signature), relative time-locks and constant number of $\wedge$ and $\vee$ in output scripts.

3.3 Preliminaries

We denote by $x \leftarrow_{\$} \mathcal{X}$ the uniform sampling of the variable x from the set $\mathcal{X}$. Throughout this paper, λ denotes the security parameter and all our algorithms run in polynomial time in λ. By writing $x \leftarrow \mathsf{A}(y)$ we mean that a *probabilistic polynomial time* algorithm A (or PPT for short) on input y, outputs x. If A is a *deterministic polynomial time* algorithm (DPT for short), we use the notation $x := \mathsf{A}(y)$. A function $\nu\colon \mathbb{N} \to \mathbb{R}$ is *negligible in* λ if for every $k \in \mathbb{N}$, there exists $n_0 \in \mathbb{N}$ s.t. for every $\lambda \geq n_0$ it holds that $|\nu(\lambda)| \leq 1/\lambda^k$. Throughout this work, we use the following notation for *attribute tuples*. Let T be a tuple of values which we call attributes. Each attribute in T is identified

using a unique keyword attr and referred to as $T.\mathsf{attr}$. Let us now briefly recall the cryptographic primitives used in this paper to establish the used notation.

A signature scheme consists of three algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, where: (i) $\mathsf{Gen}(1^n)$ gets as input 1^n and outputs the secret and public keys $(\mathsf{sk}, \mathsf{pk})$; (ii) $\mathsf{Sign}_{\mathsf{sk}}(m)$ gets as input the secret key sk and a message $m \in \{0,1\}^*$ and outputs the signature σ; and (iii) $\mathsf{Vrfy}_{\mathsf{pk}}(m;\sigma)$ gets as input the public key pk, a message m and a signature σ, and outputs a bit b. A signature scheme must fulfill correctness, i.e. it must hold that $\mathsf{Vrfy}_{\mathsf{pk}}(m; \mathsf{Sign}_{\mathsf{sk}}(m)) = 1$ for all messages m and valid key pairs $(\mathsf{sk}, \mathsf{pk})$. In this work, we use signature schemes that satisfy the notion of strong existential unforgeability under chosen message attack (or SUF–CMA). At a high level, SUF–CMA guarantees that a PPT adversary on input the public key pk and with access to a signing oracle, cannot produce a *new* valid signature on any message m.

We next recall the definition of a hard relation R with statement/witness pairs (Y, y). Let L_R be the associated language defined as $\{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a *hard relation* if the following holds: (i) There exists a PPT sampling algorithm GenR that on input 1^λ outputs a statement/witness pair $(Y, y) \in R$; (ii) The relation is poly-time decidable; (iii) For all PPT $\mathcal{A}$ the probability of $\mathcal{A}$ on input Y outputting a valid witness y is negligible.

Finally, we recall the definition of a non-interactive zero-knowledge proof of knowledge (NIZK) with online extractors as introduced in [Fis05]. The online extractability property allows for the extraction of a witness y for a statement Y from a proof π in the random oracle model and is useful for models where the rewinding proof technique is not allowed, such as UC. We need this property to prove our ECDSA-based adaptor signature scheme secure. More formally, a pair (P, V) of PPT algorithms is called a NIZK with an online extractor for a relation R, random oracle $\mathcal{H}$ and security parameter λ if the following holds: (i) *Completeness*: For any $(Y, y) \in R$, it holds that $\mathsf{V}(Y, \mathsf{P}(Y, y)) = 1$ except with negligible probability; (ii) *Zero knowledge*: There exists a PPT simulator, which on input Y can simulate the proof π for any $(Y, y) \in R$. (iii) *Online Extractor*: There exists a PPT online extractor K with access to the sequence of queries to the random oracle and its answers, such that given (Y, π), the algorithm K can extract the witness y with $(Y, y) \in R$. An instance of such a proof system is in [Fis05].

3.4 Generalized channels

3.4.1 Notation and security model

To formally model the security of generalized channels, we use the global UC framework (GUC) [CDPW07] which extends the standard UC framework [Can01] by allowing for a global setup. Here we discuss our security model (which follows the previous works on off-chain channels [DFH18, DEF+19b, DEFM19]), only briefly and refer the reader to Appendix B.1 for more details.

We consider a protocol π that runs between parties from a fixed set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{E}$ – which represents anything "external" to the current protocol execution. We assume a synchronous communication network meaning that protocol execution happens in rounds, formalized via a global ideal functionality $\widehat{\mathcal{F}}_{clock}$ representing "the clock" [KMTZ13]. Parties in the protocol are connected with authenticated communication channels with guaranteed delivery of exactly one round, formalized via an ideal functionality $\mathcal{F}_{GDC}$. For simplicity, we assume that all other communication (e.g., messages sent between the adversary and the environment) as well as local computation take zero rounds. Monetary transactions are handled by a global ideal ledger functionality $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$, where Δ is an upper bound on the blockchain delay (number of rounds it takes to publish a transaction), Σ defines the signature scheme and $\mathcal{V}$ defines valid output conditions. Furthermore, the global ledger maintains a PKI.

Generalized Channel Syntax. A *generalized channel* γ is an attribute tuple $(\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the channel identifier, $\gamma.\mathsf{users} \in \mathcal{P} \times \mathcal{P}$ defines the identities of the channel users, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ represents the total amount of coins locked in γ, and $\gamma.\mathsf{st} = (\theta_1, \ldots, \theta_n)$ is the state of γ composed of a list of *outputs*. Each output θ_i has two attributes: the value $\theta_i.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ representing the amount of coins and the function $\theta_i.\varphi\colon \{0,1\}^* \to \{0,1\}$ defining the spending condition. For convenience, we use $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{users} \to \gamma.\mathsf{users}$ defined as $\gamma.\mathsf{otherParty}(P) := Q$ for $\gamma.\mathsf{users} = \{P, Q\}$.

3.4.2 Ideal Functionality

We capture the desired functionality of a generalized channel protocol as an ideal functionality $\mathcal{F}_L$. As a first step towards defining our functionality, we informally identify the most important security and efficiency notions of interest that a generalized channel protocol should provide.

Consensus on creation: A generalized channel γ is successfully created only if all parties in $\gamma.\mathsf{users}$ agree with the creation. Moreover, parties in $\gamma.\mathsf{users}$ reach agreement on whether the channel is created or not after an a-priori bounded number of rounds.

Consensus on update: A generalized channel γ is successfully updated only if both parties in $\gamma.\mathsf{users}$ agree with the update. Moreover, parties in $\gamma.\mathsf{users}$ reach agreement on whether the update is successful or not after an a-priori bounded number of rounds.

Instant finality with punish: An honest party $P \in \gamma.\mathsf{users}$ has the guarantee that either the current state of the channel can be enforced on the ledger, or P can enforce a state where she gets all $\gamma.\mathsf{cash}$ coins. A state st is called *enforced* on the ledger if a transaction with this state appears on the ledger.

Optimistic update: If both parties in $\gamma.\mathsf{users}$ are honest, the update procedure takes a constant number of rounds (independent of the blockchain delay Δ).

Having the guarantees identified above in mind, we now design our ideal functionality $\mathcal{F}_L$. It interacts with parties from the set $\mathcal{P}$, with the adversary $\mathcal{S}$ (called the simulator) and the ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$. In a bit more detail, if a party wants to perform an action (such as opening a new channel), it sends a message to $\mathcal{F}_L$ who executes the action and informs the party about the result. The execution might leak information to the adversary who may also influence the execution which is modeled via the interaction with $\mathcal{S}$. Finally, $\mathcal{F}_L$ observes the ledger and can verify that a certain transaction appeared on-chain or the ownership of coins.

To keep $\mathcal{F}_L$ generic, we parameterized it by two values T_p and k – both of which must be independent of the blockchain delay Δ. At a high level, the value T_p upper bounds the maximal number of consecutive off-chain communication rounds between channel users. Since different parts of the protocol might require a different amount of communication rounds, the upper bound T_p might not be reached in all steps. For instance, channel creation might require more communication rounds than old state revocation. To this end, we give the power to the simulator to "speed up" the process when possible. The parameter k defines the number of ways the channel state $\gamma.\mathsf{st}$ can be published on the ledger. As discussed in Section 3.2, in this work we present a protocol realizing the functionality for $k = 1$ (see Figure 3.3). A generalized channel construction using Lightning style revocation mechanism (see Figure 3.2) would be a candidate protocol for $k = 2$.

We assume that the functionality maintains a set Γ of created channels in their latest state and the corresponding funding transaction tx. We present $\mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T_p, k)$ formally in Figure 3.4. Here we discuss each part of the functionality at a high level and argue why it captures the aforementioned security and efficiency properties identified above. We abbreviate $\mathcal{F}_L := \mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T_p, k)$.

Create. If $\mathcal{F}_L$ receives a message of the form $(\texttt{CREATE}, \gamma, \mathsf{tid}_P)$ from both parties in $\gamma.\mathsf{users}$ within T_p rounds, it expects a channel funding transaction to appear on the ledger $\mathcal{L}$ within Δ rounds. Such a transaction must spend both funding sources (defined by transaction identifiers $\mathsf{tid}_P, \mathsf{tid}_Q$) and contain one output of the value $\gamma.\mathsf{cash}$. If this is true, $\mathcal{F}_L$ stores this transaction together with the channel γ in Γ and informs both parties about the successful channel creation via the message `CREATED` (how this can be done within the UC model is discussed in Appendix B.1). Since a `CREATE` message is required from both parties and both parties receive `CREATED`, "consensus on creation" holds.

Close. Any of the two parties can request closure of the channel via the message (`CLOSE`, id), where id identifies the channel to be closed. In case both parties request closure within T rounds, *peaceful closure* is expected. This means that a transaction, spending the channel funding transaction and whose output corresponds to the latest channel state $\gamma.\mathsf{st}$, should appear on $\mathcal{L}$ within Δ rounds. If only one of the parties requests closing, $\mathcal{F}_L$ executes the `L–ForceClose` subprocedure in which case such transaction is supposed to appear on $\mathcal{L}$ within 3Δ rounds modeling possible dispute resolution. In both cases, if the

funding transaction is not spent before a certain round, an ERROR is returned to both users.

Update. The channel update is initiated by one of the parties P (called the *initiating party*) via a message $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}})$. The parameter id identifies the channel to be updated, $\overrightarrow{\theta}$ represents the new channel state and t_{stp} denotes the number of rounds needed by the parties to set up off-chain applications (e.g., new channels or fair two-party computation) that are being built on top of the channel via this update request. The update is structured into two phases: (i) the prepare phase, and (ii) the revocation phase. Intuitively, the prepare phase models the fact that both parties first agree on the new channel state and get time to set up the off-chain applications on top of this new state. The revocation phase models the fact that an update is only completed once the two parties invalidate the previous channel state. We detail the two phases in the following.

The prepare phase starts when $\mathcal{F}_L$ receives a vector of transaction identifiers $\mathsf{tid} = (\mathsf{tid}_1, \ldots, \mathsf{tid}_k)$ from $\mathcal{S}$.[2] In the optimistic case, it is completed within $3T_p + t_{\mathsf{stp}}$ rounds and ends when the initiating party P receives an UPDATE–OK message from $\mathcal{F}_L$. The setup phase can be aborted by both the initiating party P and the other party Q. This is achieved by P not sending the SETUP–OK and by Q not sending the UPDATE–OK message, respectively. This models two things. Firstly, the fact that Q might not agree with the proposed update, and secondly, that setting up off-chain objects might fail in which case parties want to abort the channel update. The abort may also result in a forceful closing of the channel via the subprocedure L–ForceClose. It happens when one of the parties has sufficient information to enforce the new state on-chain, while the other does not.

In order to complete the update, the revocation phase is executed. The functionality expects to receive the REVOKE message from both parties within $2T$ rounds, in which case $\mathcal{F}_L$ updates the channel state in Γ accordingly and informs both parties about the successful update via the message UPDATED. If one of the messages does not arrive, the subprocedure L–ForceClose is called.

To conclude, the possibility for forceful closing guarantees the security property "consensus on update" as it ensures termination of the update process and allows both parties to see the state in which the channel was closed. Moreover, in case both parties are honest, the update duration is independent of the ledger delay Δ, hence the efficiency property "optimistic update" is satisfied.

Punish. In order to guarantee "instant finality with punishments", parties continuously monitor the ledger and apply the punishment mechanism if misbehavior is detected. This is captured by the functionality in the part "Punish" which is executed at the end of each round. The functionality checks if a funding transaction of some channel was spent. If yes, then it expects one of the following to happen: (i) a punish transaction appears on $\mathcal{L}$ within Δ rounds, assigning $\gamma.\mathsf{cash}$ coins to the honest party $P \in \gamma.\mathsf{users}$; or (ii)

[2] For technical reasons, ideal functionality cannot sign transactions and thus it can also not prepare the transaction ids (which is the task of the simulator).

Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}_P) \xleftarrow{\tau_0} P$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, \mathsf{tid}_Q) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_p$: If tx s.t. $\mathsf{tx.input} = (\mathsf{tid}_P, \mathsf{tid}_Q)$ and $\mathsf{tx.output} = (\gamma.\mathsf{cash}, \varphi)$, for some φ, appears on $\mathcal{L}$ in round $\tau_1 \leq \tau + \Delta + T_p$, set $\Gamma(\gamma.\mathsf{id}) := (\{\gamma\}, \mathsf{tx})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

Wait for Q: Else wait if $(\texttt{CREATE}, \mathsf{id}) \xleftarrow{\tau \leq \tau_0 + T_p} Q$ (in that case "Both agreed" option is executed). If such message is not received, stop.

Upon $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xleftarrow{\tau_0} P$, parse $(\{\gamma\}, \mathsf{tx}) := \Gamma(\mathsf{id})$, set $\gamma' := \gamma$, $\gamma'.\mathsf{st} := \theta$:

1. In round $\tau_1 \leq \tau_0 + T_p$, let $\mathcal{S}$ define tid s.t. $|\mathsf{tid}| = k$. Then $(\texttt{UPDATE–REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{SETUP}, \mathsf{id}, \mathsf{tid}) \xhookrightarrow{\tau_1} P$.
2. If $(\texttt{SETUP–OK}, \mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{SETUP–OK}, \mathsf{id}) \xrightarrow{\tau_3 \leq \tau_2 + T_p} Q$. Else stop.
3. If $(\texttt{UPDATE–OK}, \mathsf{id}) \xleftarrow{\tau_3} Q$, then $(\texttt{UPDATE–OK}, \mathsf{id}) \xrightarrow{\tau_4 \leq \tau_3 + T_p} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop *(reject)*.
   - Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
4. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_4} P$, send $(\texttt{REVOKE–REQ}, \mathsf{id}) \xrightarrow{\tau_5 \leq \tau_4 + T_p} Q$.
   Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
5. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_5} Q$, $\Gamma(\mathsf{id}) := (\{\gamma'\}, \mathsf{tx})$, send $(\texttt{UPDATED}, \mathsf{id}, \theta) \xrightarrow{\tau_6 \leq \tau_5 + T_p} \gamma.\mathsf{users}$ and stop *(accept)*. Else set $\Gamma(\mathsf{id}) := (\{\gamma, \gamma'\}, \mathsf{tx})$, run $\texttt{L–ForceClose}(\mathsf{id})$ and stop.

Upon $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_0} P$, distinguish: **Both agreed:** If already received $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_p$, run $\texttt{L–ForceClose}(\mathsf{id})$ unless both parties are honest. In this case let $(\{\gamma\}, \mathsf{tx}) := \Gamma(\mathsf{id})$ and distinguish:

- If tx', with $\mathsf{tx}'.\mathsf{input} = \mathsf{tx.txid}$ and $\mathsf{tx}'.\mathsf{output} = \gamma.\mathsf{st}$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$, set $\Gamma(\mathsf{id}) := \bot$, send $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- Else output $(\texttt{ERROR}) \xrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop.

Wait for Q: Else wait if $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau \leq \tau_0 + T_p} Q$ (in that case "Both agreed" option is executed). If such message is not received, run $\texttt{L–ForceClose}(\mathsf{id})$ in round $\tau_0 + T_p$.

At the end of every round τ_0: For each $\mathsf{id} \in \{0,1\}^*$ s.t. $(X, \mathsf{tx}) := \Gamma(\mathsf{id}) \neq \bot$, check if $\mathcal{L}$ contains tx' with $\mathsf{tx}'.\mathsf{input} = \mathsf{tx.txid}$. If yes, then define $S := \{\gamma.\mathsf{st} \mid \gamma \in X\}$, $\tau := \tau_0 + 2\Delta$ and distinguish: **Close:** If tx'' s.t. $\mathsf{tx}''.\mathsf{input} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{output} \in S$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, set $\Gamma(\mathsf{id}) := \bot$ and $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ if not sent yet.
Punish: If tx'' s.t. $\mathsf{tx}''.\mathsf{input} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{output} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{\mathsf{pk}_P})$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, for P honest, set $\Gamma(\mathsf{id}) := \bot$, $(\texttt{PUNISHED}, \mathsf{id}) \xhookrightarrow{\tau_1} P$ and stop.
Error: Else $(\texttt{ERROR}) \xhookrightarrow{\tau} \gamma.\mathsf{users}$.

$\texttt{L–ForceClose}(\mathsf{id})$: Let τ_0 be the current round and $(X, \mathsf{tx}) := \Gamma(\mathsf{id})$. If within Δ rounds tx is still unspent on $\mathcal{L}$, then $(\texttt{ERROR}) \xrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$ and stop. *Note that otherwise, message $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \mathit{ERROR}\}$ is output latest in round $\tau_0 + 3 \cdot \Delta$.*

Figure 3.4: The ideal functionality $\mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T_p, k)$. We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$.

a transaction whose output corresponds to the latest channel state $\gamma.\mathsf{st}$ appears on $\mathcal{L}$ within 2Δ rounds, meaning that the channel is peacefully or forcefully closed. If none of the above is true, ERROR is returned. Hence, under the condition that no ERROR was returned, the security property "instant finality with punish" is satisfied.

In summary, our functionality satisfies the identified security and efficiency properties if no ERROR occurs. Otherwise, all guarantees may be lost. Hence, we are interested only in those protocols realizing $\mathcal{F}_L$ that never output an ERROR.

Notation used in the formal description in Figure 3.4. Messages sent between parties and $\mathcal{F}_L$ have the following format: (MESSAGE_TYPE, *parameters*). To shorten the description, we use the following arrow notation: by $m \overset{t}{\hookrightarrow} P$, we mean "send the message m to party P in round t." and by $m \overset{t}{\hookleftarrow} P$, we mean "receive a message m from party P in round t". To indicate that a message should be sent/received before/after a certain round, we use inequality symbols above the arrows. When $\mathcal{F}_L$ expects $\mathcal{S}$ to set certain values (such as the vector of tid's during the update process or the exact round in which a message should be sent to parties) and it does not do so, we implicitly assume that ERROR is returned. Since we do not aim to make any claims about privacy, we implicitly assume that every message that $\mathcal{F}_L$ receives/sends from/to a party is directly forwarded to $\mathcal{S}$. In the formal description, we treat the channel set Γ as a function which on input id outputs (X, tx), where X is a set of channels s.t. for every $\gamma \in X$ $\gamma.\mathsf{id} = \mathsf{id}$, if such channel exists and $\perp$ otherwise. We denote the script requiring the signature of (only) P as $\mathtt{One}\text{–}\mathtt{Sig}_{\mathsf{pk}_P}$. Moreover, we omit several natural checks that one would expect $\mathcal{F}_L$ to make. For example, messages with missing parameters should be ignored, channel instruction should be accepted only from channel users, etc. We formally define all checks as a functionality wrapper in Appendix B.6. Finally, we omit the read queries that $\mathcal{F}_L$ sends to $\mathcal{L}$ in order to learn its state (c.f. Appendix B.1).

3.5 Adaptor Signatures

Our goal is to realize the ideal functionality of generalized channels for $k = 1$, meaning that there is only one way to publish the channel state on-chain. As explained at a high level in Section 3.2.1, we achieve our goal by utilizing an adaptor signature scheme – a cryptographic primitive that we discuss in this section.

Adaptor signatures have been introduced by the cryptocurrency community to tie together the authorization of a transaction and the leakage of a secret value. An adaptor signature scheme is essentially a two-step signing algorithm bound to a secret: first, a partial signature is generated such that it can be completed only by a party knowing a certain secret, with the complete signature revealing such a secret. More precisely, we define an adaptor signature scheme with respect to a digital signature scheme Σ and a hard relation R. For any statement $Y \in L_R$, a signer holding a secret key is able to produce a *pre-signature* w.r.t. Y on any message m. Such pre-signature can be *adapted* into a valid signature on m if and only if the adaptor knows a witness for Y. Moreover, it must be possible to extract a witness for Y given the pre-signature and the adapted signature.

Despite the fact that adaptor signatures have been used in previous works (e.g. [MMS+19, Fou19, MSK]), none of these works has given a formal definition of the adaptor signature primitive and its security. In the following, we fill this gap and provide the first game-based formalization of adaptor signatures. As already mentioned, Erwig et al. [EFH+21] recently extended our definition to a two-party case.

Definition 2 (Adaptor signature scheme)**.** An adaptor signature scheme w.r.t. a hard relation R and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ consists of four algorithms $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ with the following syntax: $\mathsf{pSign}_{\mathsf{sk}}(m, Y)$ is a PPT algorithm that on input a secret key sk, message $m \in \{0,1\}^*$ and statement $Y \in L_R$, outputs a pre-signature $\tilde{\sigma}$; $\mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma})$ is a DPT algorithm that on input a public key pk, message $m \in \{0,1\}^*$, statement $Y \in L_R$ and pre-signature $\tilde{\sigma}$, outputs a bit b; $\mathsf{Adapt}(\tilde{\sigma}, y)$ is a DPT algorithm that on input a pre-signature $\tilde{\sigma}$ and witness y, outputs a signature σ; and $\mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$ is a DPT algorithm that on input a signature σ, pre-signature $\tilde{\sigma}$ and statement $Y \in L_R$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.

An adaptor signature scheme $\Xi_{R,\Sigma}$ must satisfy *pre-signature correctness* stating that for every $m \in \{0,1\}^*$ and every $(Y, y) \in R$, the following holds:

$$\Pr\left[\begin{array}{l|ll} \mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma}) = 1, & (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda), & \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y) \\ \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma) = 1, (Y, y') \in R & \sigma := \mathsf{Adapt}_{\mathsf{pk}}(\tilde{\sigma}, y), & y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y) \end{array}\right] = 1.$$

The first security property, *existential unforgeability under chosen message attack for adaptor signature* (aEUF–CMA security for short), protects the signer. It is similar to EUF–CMA for digital signatures but additionally requires that producing a forgery σ for some message m is hard even given a pre-signature on m w.r.t. a random statement $Y \in L_R$. Let us stress that allowing the adversary to learn a pre-signature on the forgery message m is crucial since, for our applications, signature unforgeability needs to hold even in case the adversary learns a pre-signature for m without knowing a witness for Y.

Definition 3 (Existential unforgeability)**.** An adaptor signature scheme, denoted as $\Xi_{R,\Sigma}$, is aEUF–CMA secure if for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ there exists a negligible function ν such that: $\Pr[\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda) = 1] \leq \nu(\lambda)$, where the experiment $\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows:

$\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$

1 : $\mathcal{Q} := \emptyset, (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
2 : $(Y, y) \leftarrow \mathsf{GenR}(1^n)$
3 : $(m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk}, Y)$
4 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
5 : $\sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma}, \mathsf{st})$
6 : **return** $\left(m \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)\right)$

$\mathcal{O}_{\mathrm{S}}(m)$

1 : $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** σ

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

1 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** $\tilde{\sigma}$

The reason why the game computes $\tilde{\sigma}$ in step 4 (although $\mathcal{A}$ could obtain it by querying $\mathcal{O}_{\mathrm{pS}}$) is that it allows $\mathcal{A}$ to learn $\tilde{\sigma}$ without m being added to $\mathcal{Q}$.

The second property, called *pre-signature adaptability*, protects the verifier. It guarantees that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be completed into a valid signature using a witness y with $(Y, y) \in R$. Notice that this property is stronger than the pre-signature correctness property from Definition 2, since we require that even pre-signatures that were not produced by pSign but are valid, can be completed into valid signatures.

Definition 4 (Pre-signature adaptability)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies pre-signature adaptability if for any message $m \in \{0,1\}^*$, any statement/witness pair $(Y, y) \in R$, any public key pk and any pre-signature $\tilde{\sigma} \in \{0,1\}^*$ with $\mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma}) = 1$, we have $\mathsf{Vrfy}_{\mathsf{pk}}(m; \mathsf{Adapt}(\tilde{\sigma}, y)) = 1$.

The last property that we are interested in is *witness extractability* which protects the signer. Informally, it guarantees that a valid signature/pre-signature pair $(\sigma, \tilde{\sigma})$ for message/statement (m, Y) can be used to extract a witness y for Y. Hence a malicious verifier cannot use a pre-signature $\tilde{\sigma}$ to produce a valid signature σ without revealing a witness for Y[3].

Definition 5 (Witness extractability)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ is *witness extractable* if for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function ν such that the following holds: $\Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda) = 1] \leq \nu(\lambda)$, where the experiment $\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows

$\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$

1 : $\mathcal{Q} := \emptyset, (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
2 : $(m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
3 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
4 : $\sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\tilde{\sigma}, \mathsf{st})$
5 : **return** $((Y, \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)) \notin R \wedge m \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma))$

$\mathcal{O}_{\mathrm{S}}(m)$

1 : $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** σ

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

1 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** $\tilde{\sigma}$

Let us stress that while the experiment $\mathsf{aWitExt}$ looks fairly similar to the experiment $\mathsf{aSigForge}$, there is one crucial difference; namely, the adversary is allowed to choose the forgery statement Y. Hence, we can assume that they know a witness for Y so they can generate a valid signature on the forgery message m. However, this is not sufficient to win the experiment. The adversary wins *only* if the valid signature does not reveal a witness for Y.

[3]We note that in order to prove security for our ECDSA-based adaptors signature scheme, the game must also check that the statement returned by the adversary is indeed sampled from the correct space, i.e., $Y \in L_R$. However, as this check is only needed for the ECDSA-based construction we did not add this restriction to the game.

Definition 6. An adaptor signature scheme $\Xi_{R,\Sigma}$ is secure, if it is aEUF–CMA secure, pre-signature adaptable, and witness extractable.

Note that none of the security definitions explicitly states that pre-signatures are unforgeable. However, it is implied by the definitions as we discuss in Appendix B.4.

3.5.1 ECDSA-based Adaptor Signature

We now construct a provably secure adaptor signature scheme based on ECDSA digital signatures that are commonly used by blockchains. The construction presented here is similar to the construction put forward by [MSK], however, some modifications are needed for the security proof. In addition to the ECDSA-based adaptor signature scheme presented here, we show a scheme based on Schnorr digital signatures, including correctness and security proofs, in Appendix B.2.

Recall the ECDSA signature scheme $\Sigma_{\mathsf{ECDSA}} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ for a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q. The key generation algorithm samples $x \leftarrow_{\$} \mathbb{Z}_q$ and outputs $g^x \in \mathbb{G}$ as the public key and x as the secret key. The signing algorithm on input a message $m \in \{0,1\}^*$, samples $k \leftarrow_{\$} \mathbb{Z}_q$ and computes $r := f(g^k)$ and $s := k^{-1}(\mathcal{H}(m) + rx)$, where $\mathcal{H}\colon \{0,1\}^* \to \mathbb{Z}_q$ is a hash function modeled as a random oracle and $f\colon \mathbb{G} \to \mathbb{Z}_q$ (i.e., f is typically defined as the projection to the x-coordinate since in ECDSA the group $\mathbb{G}$ consists of elliptic curve points). The verification algorithm on input a message $m \in \{0,1\}^*$ and a signature (r,s) verifies that $f(g^{s^{-1}\mathcal{H}(m)} X^{s^{-1}r}) = r$. One of the properties of the ECDSA scheme is that if (r,s) is a valid signature for m, then so is $(r,-s)$. Consequently, Σ_{ECDSA} does not satisfy SUF–CMA security which we need in order to prove its security. In order to tackle this problem we build our adaptor signature from the *Positive ECDSA* scheme which guarantees that if (r,s) is a valid signature, then $|s| \leq (q-1)/2$. The positive ECDSA has already been used in other works such as [BDM16, Lin21]. This slightly modified ECDSA scheme is not only assumed to be SUF–CMA but also prevents having two valid signatures for the same message after the signing process, which is useful in practice, e.g., for threshold signature schemes based on ECDSA. As the ECDSA verification accepts valid positive ECDSA signatures, these signatures can be used by any blockchain that uses ECDSA, e.g., Bitcoin.

The adaptor signature scheme in [MSK] is presented w.r.t. a relation $R_g \subseteq \mathbb{G} \times \mathbb{Z}_q$ defined as $R_g := \{(Y,y) \mid Y = g^y\}$. The main idea of the construction is that a pre-signature (r,s) for a statement Y is computed by embedding Y into the r-component while keeping the s-component unchanged. This embedding is rather involved since the value s contains a product of k^{-1}, r, and the secret key. More concretely, to compute the pre-signature for Y, the signer samples a random k and computes $K := Y^k$ and $\tilde{K} := g^k$. It then uses the first value to compute $r := f(K)$ and sets $s := k^{-1}(\mathcal{H}(m) + rx)$. To ensure that the signer uses the same value k in K and $\tilde{K}$, a zero-knowledge proof that $(\tilde{K}, K) \in L_Y := \{(\tilde{K}, K,) \mid \exists k \in \mathbb{Z}_q \text{ s.t. } g^k = \tilde{K} \wedge Y^k = K\}$ is attached to the pre-signature. We denote the prover of the NIZK as P_Y and the corresponding verifier as V_Y. The pre-signature adaptation is done by multiplying the value s with y^{-1}, where y

$\mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$	$\mathsf{pVrfy}_{\mathsf{pk}}(m, I_Y; \tilde{\sigma})$	$\mathsf{Adapt}(\tilde{\sigma}, y)$	$\mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$
$x := \mathsf{sk}, (Y, \pi_Y) := I_Y$	$X := \mathsf{pk}, (Y, \pi_Y) := I_Y$	$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$	$(r, s) := \sigma$
$k \leftarrow_{\$} \mathbb{Z}_q, \tilde{K} := g^k$	$(r, \tilde{s}, K, \pi) := \tilde{\sigma}$	$s := \tilde{s} \cdot y^{-1}$	$(\tilde{r}, \tilde{s}, K, \pi) := \tilde{\sigma}$
$K := Y^k, r := f(K)$	$u := \mathcal{H}(m) \cdot \tilde{s}^{-1}$	**return** (r, s)	$y' := s^{-1} \cdot \tilde{s}$
$\tilde{s} := k^{-1}(\mathcal{H}(m) + rx)$	$v := r \cdot \tilde{s}^{-1}$		**if** $(I_Y, y') \in R'_g$
$\pi \leftarrow \mathsf{P}_Y((\tilde{K}, K), k)$	$K' := g^u X^v$		**then return** y'
return $(r, \tilde{s}, K, \pi)$	**return** $((I_Y \in L_R)$		**else return** $\perp$
	$\wedge (r = f(K)) \wedge \mathsf{V}_Y((K', K), \pi))$		

Figure 3.5: ECDSA-based adaptor signature scheme.

is the corresponding witness for Y. This adjusts the randomness k used in s to ky, and hence matches with the r value.

Unfortunately, it is not clear how to prove security for the above scheme. Ideally, we would like to reduce both the unforgeability and the witness extractability of the scheme to the strong unforgeability of positive ECDSA. More concretely, suppose there exists a PPT adversary $\mathcal{A}$ that wins the $\mathsf{aSigForge}$ (resp. $\mathsf{aWitExt}$) experiment. Having $\mathcal{A}$, we want to design a PPT adversary (also called the simulator) $\mathcal{S}$ that breaks the $\mathsf{SUF–CMA}$ security. The main technical challenge in both reductions is that $\mathcal{S}$ has to answer queries (m, Y) to the pre-signing oracle $\mathcal{O}_{\mathrm{pS}}$ by $\mathcal{A}$. This has to be done with access to the ECDSA signing oracle but without knowledge of sk and the witness y. Thus, we need a method to "transform" full signatures into valid pre-signatures without knowing y, which seems to go against the $\mathsf{aEUF–CMA}$-security (resp. witness extractability).

Due to this reason, we slightly modify this scheme. In particular, we modify the hard relation for which the adaptor signature is defined. Let R'_g be a relation whose statements are *pairs* (Y, π), where $Y \in L_{R_g}$ is as above, and π is a non-interactive zero-knowledge proof of knowledge that $Y \in L_{R_g}$. Formally, we define $R'_g := \{((Y, \pi), y) \mid Y = g^y \wedge \mathsf{V}_g(Y, \pi) = 1\}$ and denote by P_g the prover and by V_g the verifier of the proof system for L_{R_g}. Clearly, due to the soundness of the proof system, if R_g is a hard relation, then so is R'_g.

It might seem that we did not make it any easier for the reduction to learn a witness needed for creating pre-signatures. However, we exploit the fact that we are in the ROM, and the reduction answers adversary's random oracle queries. Upon receiving a statement $I_Y := (Y, \pi)$ for which it must produce a valid pre-signature, it uses the random oracle query table to extract a witness from the proof π. Knowing the witness y and a signature (r, s), the reduction can compute $(r, s \cdot y)$ and execute the simulator of the NIZK_Y to produce a consistency proof π. This concludes the protocol description and the main proof idea. We refer the reader to Appendix B.3 for the detailed proof of the following theorem.

Theorem 2. *If the positive ECDSA signature scheme* Σ_{ECDSA} *is* $\mathsf{SUF–CMA}$*-secure and*

R_g is a hard relation, $\Xi_{R'_g,\Sigma_{\mathsf{ECDSA}}}$ from Figure 3.5 is a secure adaptor signature scheme in the ROM.

3.6 Generalized Channel Construction

We now present a concrete protocol, denoted Π_L, that requires only one commit transaction, i.e., implements the punish-then-split mechanism. This is achieved by utilizing an adaptor signature scheme $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ for signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ used by the underlying ledger and a hard relation R. Throughout this section, we assume that statement/witness pairs of R are public/secret keys of Σ. More precisely, we assume there exists a function ToKey that takes as input a statement $Y \in L_R$ and outputs a public key pk. The function is s.t. the distribution of $(\mathsf{ToKey}(Y), y)$, for $(Y, y) \leftarrow \mathsf{GenR}$, is equal to the distribution of $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}$. We emphasize that both ECDSA and Schnorr-based adaptor signatures satisfy this condition as discussed in Appendix B.5, where we also explain how to modify our protocol when this condition does not hold. Our protocol consists of four subprotocols: Create, Update, Close, and Punish. We discuss each subprotocol separately at a high level here and refer the reader to Appendix B.5 for the pseudo-code description.

Channel creation. In order to create a channel γ, the users of the channel, say A and B, have to agree on the body of the funding transaction $[\mathsf{tx}^{\mathsf{f}}]$, mutually commit to the first channel state defined by $\gamma.\mathsf{st} = ((x_A, \mathtt{One\text{–}Sig}_{\mathsf{pk}_A}), (x_B, \mathtt{One\text{–}Sig}_{\mathsf{pk}_B}))$, and sign and publish the funding transaction tx^{f} on the ledger. Recall that $\mathtt{One\text{–}Sig}_{\mathsf{pk}}$ represents the script that verifies that the transaction is correctly signed w.r.t. the public key pk. Once tx^{f} is published, the channel creation is completed. Looking at Figure 3.6, one can summarize the creation process as a step-by-step creation of transaction bodies from left to right, and then a step-by-step signature exchange on the transaction bodies from right to left. Let us elaborate on this in more detail.

Step 1: To prepare $[\mathsf{tx}^{\mathsf{f}}]$, parties need to inform each other about their funding sources, i.e., exchange the transaction identifiers tid_A and tid_B. Each party can then locally create the body of the funding transaction $[\mathsf{tx}^{\mathsf{f}}]$ with $\{\mathsf{tid}_A, \mathsf{tid}_B\}$ as input and output requiring the signature of both A and B. **Step 2:** Parties can now start committing to the initial channel state. To this end, each party $P \in \{A, B\}$ generates a *revocation public/secret* pair $(R_P, r_P) \leftarrow \mathsf{GenR}$ and *publishing public/secret* pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$, and sends the public values R_P, Y_P to the other party. Parties can now locally generate $[\mathsf{tx}^{\mathsf{c}}]$ which spends tx^{f} and can be spent by a transaction satisfying one of these conditions:

Punish A: It is correctly signed w.r.t. $\mathsf{pk}_B, \mathsf{ToKey}(Y_A), \mathsf{ToKey}(R_A)$;

Punish B: It is correctly signed w.r.t. $\mathsf{pk}_A, \mathsf{ToKey}(Y_B), \mathsf{ToKey}(R_B)$;

Channel state: It is correctly signed w.r.t. pk_A and pk_B, and at least Δ rounds have passed since tx^{c} was published.

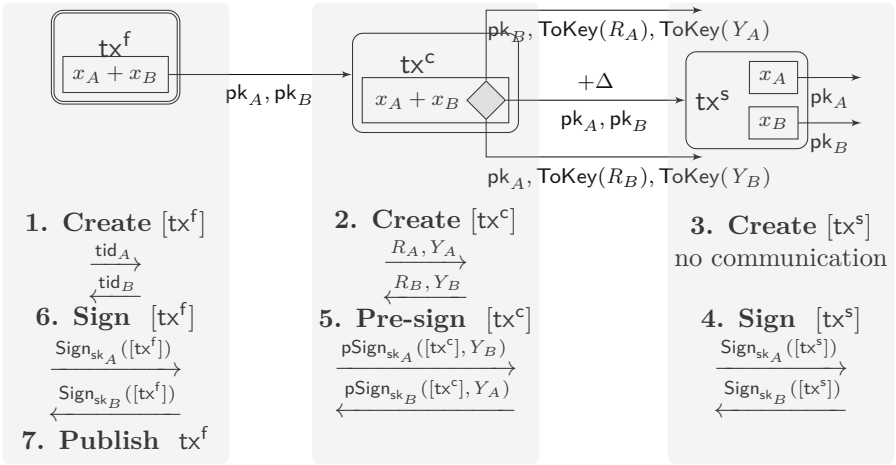

Figure 3.6: Schematic description of the generalized channel creation protocol.

Steps 3+4: Using the transaction identifier of tx^{c}, parties can generate and exchange signatures on the body of the split transaction tx^{s} which spends tx^{c} and whose output is equal to the initial state of the channel $\gamma.\mathsf{st}$. **Step 5:** Parties are now prepared to complete the committing phase by *pre-signing* the commit transaction to each other. This means that party A executes the $\mathsf{pSign}_{\mathsf{sk}_A}$ on message $[\mathsf{tx}^{\mathsf{c}}]$ and statement Y_B and sends the pre-signature to B (analogously for B). **Step 6:** If valid pre-signatures are exchanged (validity is checked using the pVrfy algorithm), parties exchange signatures on the funding transaction and post it on the ledger in **Step 7**. If the funding transaction is accepted by the ledger, channel creation is successfully completed.

The question is what happens if one of the parties misbehaves during the creation process by aborting or sending a malformed message (w.l.o.g. let B be the malicious party). If the misbehavior happens before A sends her signature on tx^{f} (i.e., before step 6), party A can safely conclude that the creation failed and does not need to take any action. If the misbehavior happens during step 6, A is in a hybrid situation. She cannot post tx^{f} on-chain as she does not have B's signature needed to spend tid_B. However, since she already sent her signature on tx^{f} to B, she has no guarantee that B will not post tx^{f} later. To resolve this issue, our protocol instructs A to spend her output tid_A. Now within Δ rounds, tid_A is spent – either by the transaction posted by A (in which case creation failed) or by tx^{f} posted by B (in which case creation succeeded).

To conclude, channel creation as described above takes 5 off-chain communication rounds, and up to Δ rounds are needed to publish the funding transaction. Our formal protocol description contains two optimizations that reduce the number of off-chain communication rounds to 3. The optimizations are based on the observations that messages sent during steps 1 and 2 can be grouped into one as well as the messages sent during steps 4 and 5.

Channel closure. The purpose of the closing procedure is to collaboratively publish the latest channel state on the blockchain. The naive implementation is to let parties

publish the latest agreed-upon commit transaction and thereafter the corresponding split transaction representing the latest channel state. However, due to the punishment mechanism built into the commit transaction, parties have to wait for Δ rounds after such a transaction is accepted by the ledger to publish the split transaction. To realize our ideal functionality, we need to design a more efficient solution eliminating the redundant waiting for honest parties.

When parties want to close a channel, they first run a "final update". In short, the final update preserves the latest channel state but removes the punishment layer. More precisely, parties agree on a new split transaction that has exactly the same outputs as the last split transaction but spends the funding transaction tx^{f} directly (i.e., **Steps 2+5** from Figure 3.6 are skipped). Once parties jointly sign the split transaction, they can publish it on the ledger which completes the channel closure. If the final update fails, parties close the channel forcefully. Namely, they first publish the latest commit transaction, wait until the time for punishment expires. Then they post the split transaction representing the final channel state. It takes at most Δ rounds to publish the commit transaction and at most 2Δ rounds to publish the split transaction once the commit transaction is accepted which corresponds to the upper bound dictated by our ideal functionality. Since forceful closing might also be triggered during a channel update (as we discuss next), we define forceful closure as a separate subprocedure `ForceClose`.

Channel update. To update a channel γ to a new state, given by a vector of output scripts θ, parties have to (i) agree on the new commit and split transaction capturing the new state and (ii) invalidate the old commit transaction.

Part (i) is very similar to the agreement on the initial commit and split transaction as described in the creation protocol (**Steps 2-5** in Figure 3.6). There is one major difference coming from the fact that the new channel state θ can contain outputs that fund other off-chain applications (such as sub-channels).[4] In order to set up these applications, the identifier of the new split transaction is needed. To this end, parties first prepare the commit (**Steps 2+3**) to learn the desired identifier and set up all applications off-chain. Once this is done, which is signaled by "`SETUP–OK`" and takes at most t_{stp} rounds, parties execute the second part of the committing phase (**Steps 4+5**).

To realize part (ii), in which the punishment mechanism of the old commit transaction is activated, parties simply exchange the revocation secrets corresponding to the previous commit transaction which completes the update. Note that in this optimistic case when both parties are honest, the update is performed entirely off-chain and takes at most $5 + t_{\mathsf{stp}}$ rounds.

We now discuss what happens if one party misbehaves during the update. As long as none of the parties pre-signed the new commit transaction, i.e., before **Step 5**, misbehavior simply implies update failure. A more problematic case is when the misbehavior occurs

[4] This is not the case during channel creation since we assume that the initial channel state consists of two accounts only.

after at least one of the parties pre-signed the new commit transaction. This happens, e.g., when one party pre-signs the new commit but the other does not; or when one party revokes the old commit and the other does not. In each of these situations, an honest party ends up in a hybrid state when the update is neither rejected nor accepted. In order to realize our ideal functionality requiring consensus on update in a bounded number of rounds, our protocol instructs an honest party to `ForceClose` the channel. This means that the honest party posts the latest commit transaction that both parties agreed on to the ledger guaranteeing that tx^{f} is spent within Δ rounds. If the transaction spending tx^{f} is the new commit transaction, the channel is closed in the updated state. Otherwise, the update fails and either the channel is closed in the state before the update, or the punishment mechanism is activated and the honest party gets financially compensated (as discussed in the next paragraph).

Punish. Since we are in the UTXO model, nothing can stop a corrupted party from publishing an old commit transaction, thereby closing the channel in an old state. However, the way we designed the commit transaction enables the honest party to punish such malicious behavior and get financially compensated. If an honest party A detects that a malicious party B posted an old commit transaction $\overline{\mathtt{TX}}_{\mathsf{c}}$, it can react by publishing a *punishment transaction* which spends $\overline{\mathtt{TX}}_{\mathsf{c}}$ and assigns all coins to A. In order to make such a punishment transaction valid, A must sign it under: (i) her secret key sk_A, (ii) B's publishing secret key $\bar{y}_B$, and (iii) B's revocation secret key $\bar{r}_B$. The knowledge of the revocation secret $\bar{r}_B$ follows from the fact that $\overline{\mathtt{TX}}_{\mathsf{c}}$ was old, i.e., parties revealed their revocation secrets to each other. The knowledge of the publishing secret $\bar{y}_B$ follows from the fact that it was B who published $\overline{\mathtt{TX}}_{\mathsf{c}}$. Let us elaborate on this in more detail. Since $\overline{\mathtt{TX}}_{\mathsf{c}}$ was accepted by the ledger, it had to include a signature of A. The only signature A provided to B on $\overline{\mathtt{TX}}_{\mathsf{c}}$ was a *pre-signature* w.r.t. $\bar{Y}_B$. The unforgeability and witness extractability properties of $\Xi_{R,\Sigma}$ guarantee that the only way B could produce a valid signature of A on $\overline{\mathtt{TX}}_{\mathsf{c}}$ was by adapting the pre-signature and hence revealing the secret key $\bar{y}_B$ to A.

Security analysis. We now formally state our main theorem, which essentially says that the Π_L protocol is a secure realization, as defined according to the UC framework, of the $\mathcal{F}_L(3,1)$ ideal functionality.

Theorem 3. *Let Σ be a* SUF–CMA *secure signature scheme, R a hard relation, and $\Xi_{R,\Sigma}$ a secure adaptor signature scheme. Let $\mathcal{L}(\Delta,\Sigma,\mathcal{V})$ be a ledger, where $\mathcal{V}$ allows for transaction authorization w.r.t. Σ, relative time-locks and constant number of Boolean operations $\wedge$ and $\vee$. Then the protocol Π_L UC-realizes the ideal functionality $\mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(3,1)$.*

The formal UC proof of the Theorem 3 can be found in Appendix B.8. Let us here just argue at a high level, why our protocol satisfies the most complex property defined by the ideal functionality, i.e., instant finality with punishment.

We first argue that instant finality holds after the channel creation, meaning that each of the two parties (alone) is able to unlock her coins from a created channel if it was

never updated. The pre-signature adaptability property of $\Xi_{R,\Sigma}$ guarantees that after a successful channel creation, each party P is able to adapt the pre-signature of the other party Q on $[\mathsf{tx}^{\mathsf{c}}]$ by using the publishing secret value y_P (corresponding to Y_P). Party P can now sign $[\mathsf{tx}^{\mathsf{c}}]$ herself and post tx^{c} on the ledger. Since parties never signed any other transaction spending tx^{f}, the posted tx^{c} will be accepted by the ledger within Δ rounds. Note that here we rely on the unforgeability of the signature scheme and the unforgeability of the adaptor signature scheme. Let us stress that parties have not revealed their revocation secrets, i.e., the values r_P and r_Q, to each other yet. The hardness of the relation R implies that none of the two parties is able to use the punishment mechanism of the published commit transaction. Thus, after Δ rounds, P can post the split transaction $\mathtt{TX}_{\mathsf{s}}$ on the ledger by which she unlocks her x_P coins.

After a successful update, each party P possesses a pre-signature of the other party Q on the new commit transaction $\mathtt{TX}_{\mathsf{c}}$ and the revocation secret of the other party on the previous commit transaction. The former implies that P is able to complete Q's pre-signature, sign $[\mathtt{TX}_{\mathsf{c}}]$ herself and post $\mathtt{TX}_{\mathsf{c}}$ on-chain. Assume first that the funding transaction of the channel tx^{f} is not spent yet, hence $\mathtt{TX}_{\mathsf{c}}$ is accepted by the ledger within Δ rounds. Since party Q does not know the revocation secret of party P corresponding to $\mathtt{TX}_{\mathsf{c}}$, by the hardness of the relation R, the only way how $\mathtt{TX}_{\mathsf{c}}$ can be spent is by publishing $\mathtt{TX}_{\mathsf{s}}$ representing the latest channel state. Hence, instant finality holds in this case.

Assume now that tx^{f} is already spent and hence $\mathtt{TX}_{\mathsf{c}}$ is rejected by the ledger. The only transaction that could have spent tx^{f} is one of the old commit transactions. This is because P never signed or pre-signed any other transaction spending tx^{f}. Let us denote the transaction spending tx^{f} as $\overline{\mathtt{TX}}_{\mathsf{c}}$. Since $\overline{\mathtt{TX}}_{\mathsf{c}}$ is an old transaction P knows Q's revocation secret r_Q. Moreover, the extractability property of the adaptor signature scheme implies that P can extract Q's publishing secret y_Q from the pre-signature that she gave to Q on this transaction and the completed signature contained in $\overline{\mathtt{TX}}_{\mathsf{c}}$. Hence, P can create a valid punishment transaction spending $\overline{\mathtt{TX}}_{\mathsf{c}}$. As our protocol instructs an honest party P to constantly monitor the blockchain and publish the punishment transaction immediately if $\overline{\mathtt{TX}}_{\mathsf{c}}$ appears on-chain, the punishment transaction will be accepted by the blockchain before the relative time-lock of $\overline{\mathtt{TX}}_{\mathsf{c}}$ expires. Hence, P receives all the coins locked in the channel which is what we needed to show.

3.7 Applications

Our generalized channels support a variety of applications such as PCNs [PD16,MMSK+17, MMS+19], payment channel hubs [TMSM21a, HAB+17], multi-path payments in PCNs [EFHR20], financially fair two-party computation [BK14], channel splitting [EMSM19], virtual payment channels [AME+21] or watchtowers [MSYS21]. Furthermore, generalized channels prove to be highly versatile in interoperable applications, i.e., applications that run across multiple blockchains (e.g., for payment channels with watchtower as described later). As generalized channels rely only on on-chain signature verification, time-locked

transactions, and basic Boolean logic, they can be implemented on a multitude of different blockchains, easing thus the design and execution of cross-chain applications. Here, we first generally discuss which applications can be built on top of generalized channels and then focus on several concrete examples.

Suitable applications. We are interested in applications that are executed among two parties (i.e., two-party applications) and whose goal is to redistribute coins between them. We call the initial transaction outputs holding coins of the two parties the *funding source* of the application. If all outputs of the funding source are contained in already published transactions, we say that the application is *funded directly by the ledger*. If the outputs are part of a generalized channel state, we say that the application is *funded by a generalized channel*.

In principle, any two-party application that can be funded directly by the underlying ledger can also be funded by a generalized channel. There are, however, two subtleties one should keep in mind. Firstly, generalized channels provide "only" instant finality with punishment. This implies that generalized channels are suitable for two-party applications in which parties are willing to accept financial compensation in exchange for an off-chain state loss. Secondly, it takes up to 3Δ rounds to publish the funding source of the application. Hence, the protocol implementing the application needs to adjust the dispute timings accordingly (if applicable). We summarize this statement in Remark 4 in Appendix B.9, where we also explain how to add applications to a generalized channel. Here we now discuss several concrete applications that benefit from generalized channels.

Fair two-party computation. One important example of an application that can be built on top of generalized channels is the *claim-or-refund* functionality introduced by Bentov and Kumaresan [BK14], and used in a series of work to realize multiple applications over Bitcoin [KB14]. At a high level, claim-or-refund allows one party, say A, to lock β coins that can be claimed by party B if she presents a witness satisfying a condition f. After a predefined number of rounds, say t, the payment of β coins is refunded back to A if the witness is not revealed.

In their work, Bentov and Kumaresan demonstrated how to utilize this simple functionality to realize a secure two-party protocol with penalties over a blockchain. Hence, the fact that claim-or-refund can be built on top of generalized channels naturally implies that two parties can execute any such protocol *off-chain*. Off-chain execution offers several advantages if both parties collaborate: (i) they do not have to pay fees or wait for the on-chain delay when deploying and funding the claim-or-refund as well as when one of the parties rightfully claims (resp. refunds) coins; (ii) they can run several simultaneous instances of claim-or-refund fully off-chain, thus improving efficiency; and (iii) a blockchain observer is oblivious to the fact that the claim-or-refund functionality has been executed off-chain. In case of misbehavior during the execution of a claim-or-refund instance, the channel punishment procedure ensures that the honest party is financially compensated with all funds locked in the channel.

Channel splitting. A generalized channel can be split into multiple sub-channels

that can be updated independently in parallel. This idea appears already in [EMSM19] where two users A and B want to split a channel γ with coin distribution (α_A, α_B) into two sub-channels γ_0 and γ_1 with the coin distributions (β_A, β_B) and $(\alpha_A - \beta_A, \alpha_B - \beta_B)$ respectively.

Executing multiple applications without prior channel splitting requires all applications to share a single funding source (i.e., that provided by the channel) and thus to be adjusted with every single channel update (i.e., even if the update is required for a single application), which might significantly increase the off-chain communication complexity. However, first splitting the channel into sub-channels effectively makes the execution of applications in each sub-channel independent of each other. For instance, two applications that benefit from channel splitting are *payment channels with watchtower* [MSYS21] and *virtual channels* [AME+21] – both of which rely on generalized channels, and which we discuss next.

Payment channels with watchtower. The security of existing channel constructions relies on the parties in a channel monitoring the blockchain to detect misbehavior. In practice, however, it is difficult to guarantee that a party remains always online. To tackle this, watchtowers [MBB+19a, ALS+18] are used as *always-online* nodes that offer monitoring services and can act on behalf of offline parties. Recently, Mirzaei et al. [MSYS21] proposed an extension to generalized channels which adds watchtower support. Their result utilizes the fact that our generalized channel construction detaches the punishment procedure from the applications.

Virtual channels. The concept of virtual channels was first introduced in the work of Dziembowski et al. [DEFM19] in which the authors presented a construction over blockchains such as Ethereum, which can run Turing complete programs. Let us shortly recall this concept. Assume Alice and Bob both have a channel with a party Ingrid, but not with each other. A virtual channel allows Alice and Bob to send off-chain payments to each other without having to communicate with Ingrid for each transaction. In a recent work, [AME+21], Aumayr et al. demonstrated that virtual payment channels are also possible over Bitcoin. Their virtual channel construction uses our generalized channels as a building block and heavily relies on the generality of our formalization. For more details, see [AME+21].

Table 3.1: Costs of lightning (LC) and generalized channels (GC) funding m HTLCs.

	on-chain (dispute)			off-chain (update)	
	# txs	size (bytes)	cost (USD)	# txs	size (bytes)
LC	$2 + m$	$513 + m \cdot 410$	$13.52 + m \cdot 10.80$	$2 + 2 \cdot m$	$706 + 2 \cdot m \cdot 410$
GC	2	663	17.47	2	$695 + m \cdot 123$

3.8 Performance Analysis

We implemented a proof of concept for our generalized channels construction, creating the necessary Bitcoin transactions. We successfully deployed these transactions on the Bitcoin testnet, demonstrating thereby compatibility with the current Bitcoin network. The source code is available at `https://github.com/generalized-channels/gc`. For the different operations, we measure the (i) number and (ii) byte size for off- and on-chain transactions required for the protocol. On-chain, we additionally measure the current estimated fee cost (May 2021). Note that the transaction fee in Bitcoin is dependent on the transaction size. We compare these numbers to Lightning-based channels.

Evaluation of multiple HTLCs. Users in a PCN typically take part in several multi-hop payments at once inside one channel. We evaluate the costs of performing m parallel payments, over both Lightning channels (LC) and generalized channels (GC). To realize multiple payments in a channel, there need to be $2 + m$ outputs: Two of which account for the balances of each user, and m representing one payment each in a "Claim-or-Refund" contract (HTLC).

To update to a channel with m parallel payments, parties need to exchange $2 + 2 \cdot m$ transactions in LC and only 2 transactions in GC. The advantage of GC is two-fold: The state is not duplicated and the HTLCs do not require an additional transaction. The difference in off-chain transaction size is $706 + 2 \cdot m \cdot 410$ bytes for LC compared to $695 + m \cdot 123$ bytes for GC.

In case of a dispute, the difference in on-chain cost is even more pronounced. To punish in LC, the honest party needs to spend $m + 1$ outputs: the one representing the balance of the malicious party and one per HTLC. This is in contrast to GC, where the honest party publishes the punishment transaction only. As a result, the total size of on-chain transactions in the LC is $513 + m \cdot 410$ bytes, which cost around $13.52 + m \cdot 10.80$ USD. In GC, the on-chain transaction size is 663 bytes resulting in a cost of 17.47 USD. There have already been disputes for channels with 50 active HTLCs [lnc20]. To settle such a dispute in LC, transactions with 21013 bytes or a cost of 553.66 USD have to be deployed. In GC, again we only need 663 bytes or 17.47 USD. GC thus reduce the on-chain cost from linear on m to constant in the case of a dispute as shown in Table 3.1.

Evaluation of channel splitting. The state duplication impacts other applications as well, e.g., channel splitting (see Section 3.7). For an LC, two commit transactions need to be exchanged per update. Hence, if we split an LC into two sub-channels, parties need to create these sub-channels for both commit transactions. Moreover, for each sub-channel two commit transactions are required. This is a total of 4 commit transactions per sub-channel. GC needs only one commitment and one split transaction per sub-channel.

After a channel split, sub-channels are expected to behave as normal channels. If we want to split an LC sub-channel further, we would need eight commit transactions (two for each of the four commitments) per sub-channel. Observe, that for every recursive

split of a channel, the amount of LC commit transactions for the new subchannel doubles. For the m^{th} split, we need 2^{m+1} additional commit transactions in the LC setting. In the GC setting, there is no state duplication, therefore the amount of transactions per sub-channel is always one commit and one split transaction. We reduce the complexity for additional transactions on the m^{th} split from exponential to constant.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program *FA 1320/1-1*, by the *DFG CRC 1119 CROSSING* (project S7), by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC), by the Austrian Science Fund (FWF) through PROFET (grant agreement P31621) and the Meitner program (grant agreement M 2608-G27), by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 projects SBA and ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by CoBloX Labs and by the ERC Project PREP-CRYPTO 724307.

CHAPTER 4

Blitz: Secure Multi-Hop Payments Without Two-Phase Commits

Abstract

Payment-channel networks (PCN) are the most prominent approach to tackling the scalability issues of current permissionless blockchains. A PCN reduces the load on-chain by allowing arbitrarily many off-chain multi-hop payments (MHPs) between any two users connected through a path of payment channels. Unfortunately, current MHP protocols are far from satisfactory. One-round MHPs (e.g., Interledger) are insecure as a malicious intermediary can steal the payment funds. Two-round MHPs (e.g., Lightning Network (LN)) follow the 2-phase-commit paradigm as in databases to overcome this issue. However, when tied with economical incentives, 2-phase-commit brings other security threats (i.e., wormhole attacks), staggered collateral (i.e., funds are locked for a time proportional to the payment path length), and dependency on specific scripting language functionality (e.g., Hash Time-Lock Contracts) that hinders a wider deployment in practice.

We present Blitz, a novel MHP protocol that demonstrates for the first time that we can achieve the best of the two worlds: a single-round MHP where no malicious intermediary can steal coins. Moreover, Blitz provides the same privacy for sender and receiver as current MHP protocols do, is not prone to the wormhole attack, and requires only constant collateral. Additionally, we construct MHPs using only digital signatures and a timelock functionality, both available at the core of virtually every cryptocurrency today. We provide the cryptographic details of Blitz and we formally prove its security. Furthermore, our experimental evaluation on a LN snapshot shows that (i) staggered collateral in LN leads to between 4x and 33x more unsuccessful payments than the constant collateral in Blitz; (ii) Blitz reduces the size of the payment contract by 26%;

and (iii) Blitz prevents up to 0.3 BTC (3397 USD in October 2020) in fees being stolen over a three day period as it avoids wormhole attacks by design.

This chapter presents the results of a collaboration with Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei, which was published at the 30th USENIX Security Symposium in 2021 under the title "Blitz: Secure Multi-Hop Payments Without Two-Phase Commits". I am the main author of this paper. I am responsible for the idea, designing and writing the protocol, formalization in the UC framework, defining the security and privacy properties, writing the proofs, writing a proof-of-concept implementation, conducting the experiments and the simulation, as well as comparison to related work. Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei were the general advisors and contributed with continuous feedback.

4.1 Introduction

Permissonless cryptocurrencies such as Bitcoin enable secure payments in a decentralized, trustless environment. Transactions are verified through a consensus mechanism and all valid transactions are recorded in a public, distributed ledger, often called blockchain. This approach has inherent scalability issues and fails to meet the growing user demands: In Bitcoin, the transaction throughput is technically limited to tens of transactions per second and the transaction confirmation time is around an hour. In contrast, more centralized payment networks such as the Visa credit card network, can handle peaks of 47,000 transactions per second.

This scalability issue is an open problem in industry and academia alike [GMSR+20, ZABZ+21]. Among the approaches proposed so far, payment channels (PC) have emerged as one of the most promising solutions; implementations thereof are already widely used in practice, e.g., the Lightning Network (LN) [PD16] in Bitcoin. A PC enables two users to securely perform an arbitrary amount of instantaneous transactions between each other while burdening the blockchain with merely two transactions, (i) for opening and (ii) for closing. In particular, following the unspent transaction output (UTXO) model, two users open a PC by locking some coins in a shared multi-signature output. By exchanging signed transactions that spend from the shared output in a peer-to-peer fashion, they can capture and redistribute their balances off-chain. Either one of the two users can terminate the PC by publishing the latest of these signed transactions on the blockchain.

As creating PCs requires locking up some coins, it is economically infeasible to set up a PC with every user one wants to interact with. Instead, PCs can be linked together forming a graph known as payment channel network (PCN) [PD16, MMSK+17]. In a PCN, a payment of α coins from a sender U_0 to a receiver U_n can be performed via a path $\{U_i\}_{i\in[0,n]}$ of intermediaries.

4.1.1 State-of-the-art PCNs

A possible way of achieving such a multi-hop payment (MHP) is an optimistic 1-round approach, e.g., Interledger [TS15]. Here, U_0 starts paying to its neighbor on the path U_1, who then pays to its neighbor U_2 and so on until U_n is reached. This protocol, however, relies on every intermediary behaving honestly, otherwise any intermediary can trivially steal coins by not forwarding the payment to its neighbor.

To achieve security in MHPs, most widely deployed PCNs (e.g., LN [PD16]) require an additional second round of communication (i.e., sequential, pair-wise communication between sender and receiver via intermediaries). Specifically, PCNs follow the principles of the 2-phase-commit protocol used to perform atomic updates in distributed databases. In the first communication round, the users on the payment path lock α coins of the PC with their right neighbor in a simple smart contract called Hash Time-Lock Contract (HTLC), which can be expressed even in restricted scripting languages such as the one used in Bitcoin. The money put into the HTLC by the left neighbor at each PC moves to the right neighbor, if this neighbor can present a secret chosen by U_n (i.e., the receiver of the payment); alternatively, it can be reclaimed by the left neighbor after some time has expired.

After HTLCs have been set up on the whole path, the users move to the second round, where they release the locks by passing the secret from U_n to U_0 via the intermediaries on the path before the time on the HTLCs has expired. Intermediaries are economically incentivized to assist in the 2-phase payment protocol. In the first round, when U_i receives α coins from the left neighbor U_{i-1}, it forwards only $\alpha - \mathsf{fee}$ to the right neighbor U_{i+1}, charging fee coins for the forwarding service. In the second round, when U_{i+1} claims the $\alpha - \mathsf{fee}$ coins from U_i, the latter is incentivized to recover the α coins from U_{i-1}.

4.1.2 Open problems in current PCNs

There are some fundamental problems with current PCNs that follow the 2-phase-commit paradigm. While 2-phase-commit has been successfully used for atomic updates in distributed databases, it is not well suited to applications where economic incentives are inherently involved. In particular, there exists a tradeoff between security, efficiency, and the number of rounds in the PCN setting that constitutes not only a challenging conceptual problem but also one with strong practical impact, as we motivate below.

Staggered collateral. After a user U_i has paid to U_{i+1}, it must have enough time to claim the coins put by U_{i-1}. If U_{i-1} is not cooperative, then this time is used to forcefully claim the funds with an on-chain transaction. The timing on the HTLCs (called collateral time in the blockchain folklore) grows therefore in a staggered manner from right to left, $t_i \geq t_{i+1} + \xi$. In practice, ξ has to be quite long: e.g., in the LN, it is set to one day (144 blocks). In the worst case, the funds are locked up for a time of $n \cdot \xi$. This means that a single payment of value α over n users can lock up a collateral of $\Theta(n^2 \cdot \alpha \cdot \xi)$. Reducing this locktime enables a faster release time of locked funds and directly improves the

throughput of the network. Moreover, long locktimes are also problematic when looking at the high volatility of cryptocurrency prices, where prices can drop significantly within the same day.

Griefing attack. A malicious user can start an MHP to itself, causing user U_i to lock up α coins for a time $(n-i)\cdot\xi$. The malicious user subsequently goes idle and lets the payment fail with the intention of reducing the overall throughput of the network by causing users to lock up their funds. In a different scenario, an intermediary could do the same by accepting payments in the first round but going idle in the second. It is interesting also to observe the amplification factor: with the relatively small amount of α coins, an attacker can lock $(n-1)\cdot\alpha$ coins of the network. This attack is hard to detect and can even be used to target specific users in the PCN in order to lock up their funds.

Wormhole attack. The wormhole attack [MMS+19] is an attack on PCNs where two colluding malicious users skip honest users in the open phase of the 2-phase-commit protocol and thereby cheat them out of their fees. The payment does not happen atomically anymore: For some users the payment is successful and for others, it is not, i.e., for the ones encased by the malicious users. The users for whom it is unsuccessful have to lock up some of their funds, but do not get any fees for offering their services, nor can they use their locked funds for other payments. These fees go instead to the attacker.

HTLC contracts. PCNs built on top of 2-phase-commit payments depend largely on HTLCs and the underlying cryptocurrencies supporting them in their scripts. However, there are a number of cryptocurrencies that do not have this functionality or that do not provide scripting capabilities at all, such as Stellar or Ripple. Instead, these currencies provide only digital signature schemes and timelocks.

On a conceptual level, one could actually wonder whether or not it is required to add an agreement protocol (in the database literature, a protocol where if an honest party delivers a message m, then m is eventually delivered by every honest party), like the HTLC-based 2-phase-commit paradigm, on top of the blockchain-inherited consensus protocol.

The current state of affairs thus leads to the following question: *Is it possible to design a PCN protocol with a single round of communication (and thus without HTLCs) while preserving security and atomicity?*

4.1.3 Our contributions

We positively answer this question by presenting *Blitz*, a novel payment protocol built on top of the existing payment channel constructions, which combines the advantages of both the optimistic 1-round and the 2-phase-commit paradigms. Our contributions are as follows.

- With Blitz, we introduce for the first time a payment protocol that achieves an MHP in one round of communication while preserving security in the presence of malicious

intermediaries (i.e., as in the LN). The Blitz protocol has constant collateral of only $\Theta(n \cdot \alpha \cdot \xi)$, allowing for PCNs that are far more robust against griefing attacks and provide a higher transaction throughput. Additionally, the Blitz protocol is immune to the wormhole attack, and having only one communication round reduces the chance of unsuccessful payments due to network faults.

- We show that Blitz payments can be realized with only timelocks and signatures, without requiring, in particular, HTLCs. This allows for a more widespread deployment, i.e., in cryptocurrencies that do not feature hashlocks or scripting, but only signatures and timelocks, e.g., Stellar or Ripple. Since Blitz builds on standard payment channel constructions, it can be smoothly integrated as an (alternative or additional) multi-hop protocol into all popular PCNs, such as the LN.

- We formally analyze the security and privacy of Blitz in the Universal Composability (UC) framework. We provide an ideal functionality modeling the security and privacy notions of interest and show that Blitz is a UC realization thereof.

- We evaluate Blitz and show that while the computation and communication overhead is inline with that of the LN, the size of the contract used in Blitz is around 26% smaller than an HTLC in the LN, which in practice opens the door for a higher number of simultaneous payments within each channel. We have additionally evaluated the effect of the reduction of collateral from staggered in the LN to constant in Blitz and observed that it reduces the number of unsuccessful payments due to locked funds by a factor between 4x and 33x, depending on payment amount and percentage of disrupted payments. Finally, the avoidance of the wormhole attack by design in Blitz can save up to 0.3 BTC (3397 USD in October 2020) of fees in our setting (over a three day period).

4.2 Background and notation

The notation used in this work is adopted from [AEE+21]. We provide here an overview of the necessary background and for more details we refer the reader to [MMS+19, MMSK+17, AEE+21].

4.2.1 Transactions in the UTXO model

Throughout this work, we consider cryptocurrencies that are built with the *unspent transaction output* (UTXO) model, as Bitcoin is for instance. In such a model, the units of cash, which we will call *coins*, exist in *outputs* of *transactions*. Let us define such an output θ as a tuple consisting of two values, $\theta := (\mathsf{cash}, \phi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins held in this output and $\theta.\phi$ is the condition which must be fulfilled in order to spend this output. The condition is encoded in the scripting language used by the underlying cryptocurrency. We say that a user U owns the coins in an output θ, if $\theta.\phi$ contains a digital signature verification script w.r.t. U's public key and the digital signature scheme of the underlying cryptocurrency. For this, we use the notation $\mathsf{OneSig}(U)$. If multiple signatures are required, we write $\mathsf{MultiSig}(U_1, \ldots, U_n)$.

Figure 4.1: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx′ has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown.

Ownership of outputs can change via transactions. A transaction maps a non-empty list of existing outputs to a non-empty list of new outputs. For better distinction, we refer to these existing outputs as *transaction inputs*. We formally define a transaction body tx as an attribute tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. The identifier $\mathsf{tx.id} \in \{0,1\}^*$ is automatically assigned as the hash of the inputs and outputs, $\mathsf{tx.id} := \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$, where $\mathcal{H}$ is modelled as a random oracle. The attribute tx.input is a list of identifiers of the inputs of the transaction, while $\mathsf{tx.output} := (\theta_1, \ldots, \theta_n)$ is a list of new outputs. A full transaction $\overline{\mathsf{tx}}$ contains additionally a list of witnesses, which fulfill the spending conditions of the inputs. We define $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience $\overline{\mathsf{tx}} := (\mathsf{tx}, \mathsf{witness})$. Only a valid transaction can be published on the blockchain, i.e., one that has a valid witness for every input and has only inputs not used in other published transactions.

In fact, a transaction is not published on the blockchain immediately after it is submitted, but only after it is accepted through the consensus mechanism. We model that by defining a blockchain delay Δ, an upper bound on the time it takes for a transaction that is broadcast until it is added to the ledger.

For better readability, we use charts to visualize transactions, their ordering, and how they are used in protocols. The charts are expected to be read from left to right, i.e., the direction of the arrows. Every transaction is represented as a rectangle with rounded corners. Incoming arrows represent inputs. Every transaction has one or more output boxes inside it. Inside these boxes, we write the amount of coins stored in the corresponding output. Every output box has one or more outgoing arrows. This arrow has the condition needed to spend the corresponding output written above and below it.

To present complex conditions in a compact way, we use the following notation. On a high level, we write the owner(s) of an output below the arrow and how they can spend it above. In a bit more detail, most output scripts require signature verification w.r.t. one or more public keys, a condition that we represent by writing the necessary public keys below a given arrow. Other conditions are written above the arrow. The conditions above can be any script supported by the underlying cryptocurrency, however, in this paper, we require only the following. We write "$+t$" or $\mathsf{RelTime}(t)$ to denote a relative timelock, i.e., the output with this condition can be spent, if and only if at

least t rounds have passed since the transaction containing the output was published on the blockchain. Additionally, we consider absolute timelocks, denoted as "$\geq t$" or $\mathsf{AbsTime}(t)$: this condition is satisfied if and only if the blockchain is at least t blocks long. If an output condition is a disjunction of several conditions, i.e., $\phi = \phi_1 \vee \cdots \vee \phi_n$, we write a diamond shape in the output box and put each subcondition ϕ_i above/below its own arrow. For the conjunction of several conditions, we write $\phi = \phi_1 \wedge \cdots \wedge \phi_n$. We illustrate an example of our transaction charts in Figure 4.1.

4.2.2 Payment channels

A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions (for opening and closing). The balances are kept and updated in what is called a state. For brevity and readability, we hereby abstract away from the implementation details of a payment channel and provide a more detailed description in Appendix C.3.

We assume that there is an off-chain transaction $\mathsf{tx}^{\mathsf{state}}$ which holds the outputs representing the current state of the payment channel. We further assume that the current $\mathsf{tx}^{\mathsf{state}}$ can always be published on the blockchain and if an old state is published by a dishonest user, the honest user gets the total channel balance through some punishment mechanism.

Formally, we define a channel $\overline{\gamma}$ as the following attribute tuple $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$. Here, $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ is a unique identifier of the channel, $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ denotes the two parties that participate in the channel out of the set of all parties $\mathcal{P}$. Further, $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ stores the total number of coins held in the channel and $\overline{\gamma}.\mathsf{st} := (\theta_1, \ldots, \theta_n)$ is the current state of the channel consisting of a list of outputs. For convenience, we also define a channel skeleton γ with respect to a channel $\overline{\gamma}$ as the tuple $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$. When the channel is used along a payment path as shown in the next section, we say the $\gamma.\mathsf{left} \in \gamma.\mathsf{users}$ accesses the user that is closer to the sender and $\gamma.\mathsf{right} \in \gamma.\mathsf{users}$ the one closer to the receiver. The balance of each user can be inferred from the state $\overline{\gamma_i}.\mathsf{st}$, however for convenience we define a function $\overline{\gamma_i}.\mathsf{balance}(U)$, that returns the coins of user $U \in \gamma_i.\mathsf{users}$ in this channel.

4.2.3 Payment channel networks

Since maintaining a payment channel locks a certain amount of coins for a party, it is economically prohibitive to set up a payment channel with every party that one potentially wants to interact with. Instead, each party may open channels with a few other parties, creating thereby a network of channels. A payment channel network (PCN) [MMSK+17] is thus a graph where vertices represent the users and edges represent channels between pairs of users. In a PCN, a user can pay any other user connected through a path of payment channels between them. Suppose user U_0 wants to pay some amount α to U_n, but does not have a payment channel directly with it. Now assume that instead, U_0 has a payment channel $\overline{\gamma_0}$ with U_1, who in turn has a channel $\overline{\gamma_1}$ with U_2 and so on, until

the receiver U_n. We say that U_0 and U_n are connected by a path and denote a payment using it as *multi-hop payment* (MHP).

Optimistic payment schemes. In an MHP, the main challenge is to ensure that the payment happens atomically and for everyone, so that no (honest) user loses any money. In fact, there exist payment-channel network constructions where this security property does not hold. We call them *optimisic payment schemes* and give Interledger [TS15] as an example. In this scheme, the users on the path simply forward the payment without any guarantee of the payment reaching the receiver. The sender U_0 starts by performing an update for channel $\overline{\gamma_0}$, where $\overline{\gamma_0}.\mathsf{balance}(U_1)$ is increased by α (and $\overline{\gamma_0}.\mathsf{balance}(U_0)$ is decreased by α) compared to the previous state. U_1 does the same with U_2 and this step is repeated until the receiver U_n is reached. This scheme works if every user is honest. However, a malicious intermediary can easily steal the money by simply stopping the payment and keeping the money for itself.

Secure MHPs. Since the assumption that every user is honest is infeasible in practice, most widely deployed systems instead ensure that no honest user loses coins. The Lightning Network (LN) [PD16] uses so-called Hash Time-Lock Contracts (HTLCs). An HTLC works as follows. In a payment channel between Alice and Bob, party Alice locks some coins that belong to her in an output that is spendable in the following fashion: (i) After a predefined time t, Alice gets her money back. (ii) Bob can also claim the money at any time if he knows a pre-image r_A for a certain hash value $\mathcal{H}(r_A)$, which is set by Alice.

For an MHP in the LN, suppose again that we have a sender U_0 who wants to pay α to a receiver U_n via some intermediaries U_i with $i \in [1, n-1]$, and that two users U_j and U_{j+1} for $j \in [0, n-1]$ have an opened payment channel. Now for the first step, U_n samples a random number r, computes the hash of it $y := \mathcal{H}(r)$, and sends y to U_0. In the second step, the sender U_0 sets up an HTLC with U_1 by creating a new state with three outputs $\theta_1, \theta_2, \theta_3$ that correspondingly hold the amount of coins: α, U_0's balance minus α and U_1's balance. While θ_2 and θ_3 are spendable by their respective owners, θ_1 is the output used by the HTLC. The HTLC that is constructed spends the output containing α back to U_0 after n time, let us say n days, or to U_1 if it knows a value x such that $\mathcal{H}(x) = y$. Now U_1 repeats this step with its right neighbor, again using y but a different time, $(n-1)$ days, in the HTLC. This step is repeated until the receiver is reached, with a timeout of 1 day.

Now if constructed correctly, the receiver U_n can present r to its left neighbor U_{n-1}, which is the secret required in the HTLC for giving the money to U_n. We call this *opening the HTLC*. After doing that, the two parties can either agree to update their channel to a new state, where U_n has α coins more, or otherwise the receiver can publish the state and a transaction with witness r spending the money from the HTLC to itself on-chain. When a user U_i reveals the secret r to its left neighbor U_{i-1}, U_{i-1} can use r to continue this process. For this continuation, U_{i-1} needs to have enough time. Otherwise, U_i could claim the money of the HTLC it has with U_{i-1} by spending the HTLC on-chain at the

last possible moment. Because of the blockchain delay, user U_{i-1} will notice this too late and will not be able to claim the money of the HTLC with U_{i-2} anymore. This is the reason why the timelocks on the HTLCs are staggered, i.e., increasing from right to left.

The aforementioned process where each user presents r to the left neighbor is repeated until the sender U_0 is reached, at which point the payment is completed. We call this approach of performing MHPs *2-phase-commit*.

4.3 Solution overview

The goal of this work is to achieve the best of the two multi-hop payment (MHP) paradigms existing nowadays (optimistic and 2-phase-commit), that is, an MHP protocol with a single round of communication that overcomes the drawbacks of the current LN MHP protocol and yet maintains the security and privacy notions of interest.

For that, we propose a paradigm shift, which we call *pay-or-revoke*. The idea is to update the payment channels from sender to receiver in a single round of communication. The key technical challenge is thus to design a single channel update that can be used *simultaneously* for sending coins from the left neighbor to the right one if the payment is successful and for a refund of the coins to the left neighbor if the payment is unsuccessful (e.g., one intermediary is offline).

We present the pay-or-revoke paradigm in an incremental way, starting with a naive design, discussing the problems with it, and presenting a tentative solution. We iterate these steps until we finally reach our solution.

Naive approach. Assume a setting with a sender U_0 who wants to pay α coins to a receiver U_n via a known path of some intermediaries U_i ($i \in [1, n-1]$), where each pair of consecutive users U_j and U_{j+1} for $j \in [0, n-1]$ has a payment channel $\overline{\gamma_j}$, where $\overline{\gamma_j}.\mathsf{balance}(U_j) \geq \alpha$. We start out with an optimistic payment scheme, as presented in Section 4.2.3. We already explained that the success of such a payment relies on every intermediary behaving honestly and really forwarding the α coins. Should an intermediary not forward the payment, U_n will never receive anything. Additionally, a receiver could claim that it never received the money even though it actually did and it would be difficult for the sender to prove otherwise.

To solve these problems the sender faces when using this form of payment we introduce a possibility for the sender to step back from a payment, that is, refund itself and all subsequent users the α coins that they initially put, should the payment not reach U_n. With such a refund functionality, the sender can now check if a receiver is giving a confirmation that it got the payment. This confirmation is external to the system (e.g., a digital payment receipt) and serves additionally as a proof that the money was received. If such a confirmation is not received, the sender simply steps back from the payment, and the payments in every channel are reverted.

Adding refund functionality. Adding a refund functionality while avoiding additional security problems is challenging. Two neighbors can no longer simply update their

channel $\overline{\gamma_i}$ to a state where α coins are moved from the left to the right neighbor, as this only encodes the payment. Instead, we need to introduce an intermediate channel state $\mathsf{tx}^{\mathsf{state}}$, which encodes the possibility for both a refund and a payment.

We realize that as follows. This new state has an output holding α coins coming from $\overline{\gamma_i}.\mathsf{left}$ $(= U_i)$ while leaving the rest of the balance in the channel untouched. The output containing α coins becomes then the input for two mutually exclusive transactions: refund and payment. We denote the refund transaction as $\mathsf{tx}_i^{\mathsf{r}}$, which spends the money back to $\overline{\gamma_i}.\mathsf{left}$ $(= U_i)$. We denote the payment transaction as $\mathsf{tx}_i^{\mathsf{p}}$, which spends the money to $\overline{\gamma_i}.\mathsf{right}$ $(= U_{i+1})$. The refund should only be possible until a certain time T. This gives the sender time to wait for the payment to reach the receiver and for the receiver to give a (signed) confirmation. Should something go wrong, the sender starts the refund procedure. After time T, if no refund happened, the payment is considered successful and the payment transaction becomes valid.

The latter condition can easily be expressed in the scripting language of virtually any cryptocurrency including Bitcoin, by making use of absolute timelocks, which in this work we defined as $\mathsf{AbsTime}(T)$, meaning an output can be spent only after some time T. Unfortunately, the same cannot be done for expressing the condition that an output is spendable *only before* time T (e.g., see [EMSM19] for details).

We overcome this problem in a different way. Instead of making the refund transaction $\mathsf{tx}_i^{\mathsf{r}}$ only valid before T, we allow both $\mathsf{tx}_i^{\mathsf{r}}$ and the payment transaction $\mathsf{tx}_i^{\mathsf{p}}$ to be valid after time T and encode a condition that, should both be posted after T, $\mathsf{tx}_i^{\mathsf{p}}$ will always be accepted over $\mathsf{tx}_i^{\mathsf{r}}$. We can achieve this by adding a relative timelock on the input of $\mathsf{tx}_i^{\mathsf{r}}$ of the blockchain delay Δ. In other words, should a user try to close the channel with $\mathsf{tx}^{\mathsf{state}}$ appearing on the chain after time T, the other user will have enough time to react and post $\mathsf{tx}_i^{\mathsf{p}}$, which will get accepted before the relative timelock of $\mathsf{tx}_i^{\mathsf{r}}$ expires. For the honest refund case nothing changes: If $\mathsf{tx}^{\mathsf{state}}$ is on-chain and $\mathsf{tx}_i^{\mathsf{r}}$ gets posted before $T - \Delta$, it will always be accepted over $\mathsf{tx}_i^{\mathsf{p}}$, since the latter transaction is only valid after time T.

Making the refund atomic. So far, we added a refund functionality that is (i) not atomic and (ii) triggerable by every user on the path. An obvious attack on this scheme would be for any user on the path to commence the refund in a way that $\mathsf{tx}_i^{\mathsf{r}}$ is accepted on the ledger just before T. Other users would not have enough time to react accordingly and lose their funds. Also, allowing intermediary users to start the refund opens up the door to griefing, where malicious users start a refund even though the payment reached the receiver. We therefore need a mechanism that (i) ensures the atomicity of the refund (or payment) and (ii) is triggerable only by the sender.

Following the LN protocol, one could add a condition $\mathcal{H}(r_A)$ on the refund transaction, such that the refund can only happen when a pre-image r_A chosen by the sender is known. To prevent the sender from publishing at the last moment however, the timing for the refund in the next channel would have to be $T + \Delta$ to give U_1 enough time to react. In subsequent channels, this time would grow by Δ for every hop and we would

then have an undesirable staggered time delay. Additionally, this approach would rely on the scripting language supporting hash-lock functionality.

To keep the time delay constant, we instead make the refund transactions dependent on a transaction being published by the sender. First, the sender creates a transaction that we name *enable-refund* and denote by $\mathsf{tx}^{\mathsf{er}}$. The unsigned transaction $\mathsf{tx}^{\mathsf{er}}$ is then passed through the path and is used at each channel $\overline{\gamma_j}$ as an additional input for $\mathsf{tx}^{\mathsf{r}}_i$.

This makes the refund transaction at every channel dependent on $\mathsf{tx}^{\mathsf{er}}$ and gives the sender and only the sender the possibility to abort the payment until time T in case something goes wrong along the path (e.g., a user is offline or the enable-refund transaction is tampered), and the receiver the guarantee to get the payment after time T otherwise.

In order to use the same $\mathsf{tx}^{\mathsf{er}}$ for the refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ of every channel $\overline{\gamma_i}$, we proceed as follows. For every user on the path (except for the receiver) there needs to exist an output in $\mathsf{tx}^{\mathsf{er}}$ that belongs to it. Additionally, we observe that an intermediary U_i whose left neighbor U_{i-1} has used $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ can safely construct a refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ dependent on the same $\mathsf{tx}^{\mathsf{er}}$, because it will know that if its left neighbor refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be on-chain, which means that it can refund itself. Also, since the appearance of $\mathsf{tx}^{\mathsf{er}}$ on the ledger is a global event that is observable by everyone at the same time, the time T used for the refund can be the same for every channel, i.e., constant.

Putting everything together. Our approach is depicted in Figure 4.2, $\mathsf{tx}^{\mathsf{er}}$ is shown in Figure 4.3, and the transaction structure between two users is shown in Figure 4.4. Note that we change the payment value from α to α_i to embed a per-hop fee (see Appendix C.1 for details). After the payment is set up from sender to receiver, the receiver sends a confirmation of $\mathsf{tx}^{\mathsf{er}}$ back to U_0, which acts both as verification that $\mathsf{tx}^{\mathsf{er}}$ was not tampered and as a payment confirmation. Should the sender receive this in time, it will wait until time T, after which the payment will be successful. If no confirmation was received in time, or $\mathsf{tx}^{\mathsf{er}}$ was tampered with, the sender will publish $\mathsf{tx}^{\mathsf{er}}$ in time to trigger the refund.

We remark that it is crucial that every intermediate user can safely construct $\mathsf{tx}^{\mathsf{r}}_i$ only observing $\mathsf{tx}^{\mathsf{er}}$, but not the input funding it (or not even knowing whether it will be funded at all in the first place). Indeed, an intermediary U_i does not care if the transaction $\mathsf{tx}^{\mathsf{er}}$ is spendable at all, it only cares that its left neighbor U_{i-1} uses an output of the same transaction $\mathsf{tx}^{\mathsf{er}}$ as input for its refund transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$, as U_i does in $\mathsf{tx}^{\mathsf{r}}_i$.

In UTXO-based cryptocurrencies, using the j^{th} output of a transaction tx as input of another transaction tx' means referencing the hash of the transaction body $\mathcal{H}(\mathsf{tx})$, which we defined as $\mathsf{tx.id}$, plus an index j. A transaction $\mathsf{tx}^{\mathsf{r}}_i$ that was created with an input referencing $\mathsf{tx}^{\mathsf{er}}.\mathsf{id}$ and some index j, can only be valid if $\mathsf{tx}^{\mathsf{er}}$ is published. This means, in particular, that it is computationally infeasible to create a different transaction $\mathsf{tx}^{\mathsf{er}\prime} \neq \mathsf{tx}^{\mathsf{er}}$ and use one of $\mathsf{tx}^{\mathsf{er}\prime}$'s outputs as input of $\mathsf{tx}^{\mathsf{r}}_i$ without finding a collision in $\mathcal{H}$. Further, as $\mathsf{tx}^{\mathsf{r}}_i$ requires the signatures of both U_i and U_{i+1}, a malicious U_i on its own cannot create a different refund transaction $\mathsf{tx}^{\mathsf{r}\prime}_i$ that does not depend on $\mathsf{tx}^{\mathsf{er}}$.

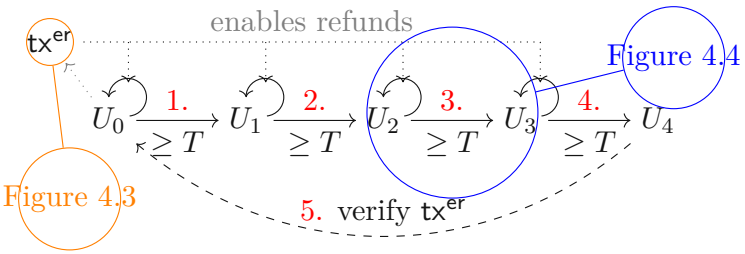

Figure 4.2: Illustration of the pay-or-revoke paradigm.

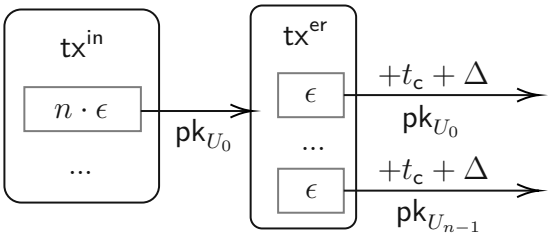

Figure 4.3: Transaction $\mathsf{tx}^{\mathsf{er}}$, which enables the refunds and, here, spends the output of some other transaction $\mathsf{tx}^{\mathsf{in}}$.

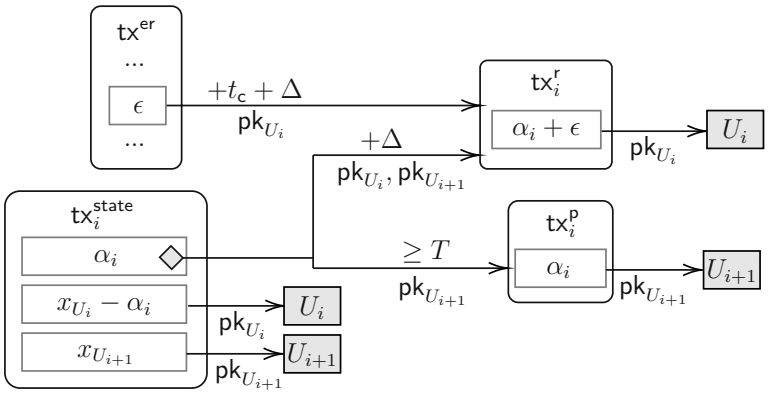

Figure 4.4: Payment setup in the channel $\overline{\gamma_i}$ of two neighboring users U_i and U_{i+1} with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{U_i} and $x_{U_{i+1}}$ are the amounts that U_i and U_{i+1} own in the state prior to $\mathsf{tx}^{\mathsf{state}}$.

A final timelock. There is however still one subtle problem with the construction up to this point regarding the timing coming from the fact that the sender has the advantage of being the only one able to trigger the refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In a bit more detail, as closing a channel takes some time, a malicious sender U_0 can forcefully close its channel with U_1 beforehand. Then, when $\mathsf{tx}_0^{\mathsf{state}}$ is on the ledger, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ so that it appears just before $T - \Delta$. The sender is able to publish $\mathsf{tx}_0^{\mathsf{r}}$ just in time before T. All other intermediaries, however, who did not yet close their channel, with the result that

$\mathsf{tx}_i^{\mathsf{state}}$ is not on the ledger, will not be able to do this and publish $\mathsf{tx}_i^{\mathsf{r}}$ in time.

To solve this problem, we introduce a relative timelock on the outputs of $\mathsf{tx}^{\mathsf{er}}$ of exactly $t_{\mathsf{c}} + \Delta$, as shown in Figure 4.3 and Figure 4.4. This relative time delay is an upper bound on the time it takes to (i) forcefully close the channel and (ii) wait for the time delay needed to publish $\mathsf{tx}_i^{\mathsf{r}}$. With this, we ensure that no user gains an advantage by closing its channel in advance, since this can be entirely done in this relative timelock on $\mathsf{tx}^{\mathsf{er}}$'s outputs. Honest intermediaries can easily check that this relative timelock is present in $\mathsf{tx}^{\mathsf{er}}$'s outputs and every user on the payment path has the same time.

A timeline of when the transactions have to appear on the ledger is given in Appendix C.6. Note that for the payment to be refunded, $\mathsf{tx}^{\mathsf{er}}$ has to be posted to the ledger at the latest at time $T - t_{\mathsf{c}} - 3\Delta$. Still, for better readability, we sometimes refer to this case simply as $\mathsf{tx}^{\mathsf{er}}$ being published before time T.

Improving anonymity of the path. Until this point, we have shown a design of the pay-or-revoke paradigm, that, while ensuring that honest users do not lose coins, has an obvious drawback in terms of anonymity. In particular, the transaction outputs of $\mathsf{tx}^{\mathsf{er}}$ contain the addresses of every user on the path in the clear (except for the receiver who does not need to refund and therefore needs no such output). This means that every intermediary (or any other user that sees $\mathsf{tx}^{\mathsf{er}}$) learns about the identity of every user on the payment path as soon as it sees $\mathsf{tx}^{\mathsf{er}}$. To prevent this leak, we use stealth addresses [VS18]. We overview our use of stealth addresses here and refer to Section 4.4.2 for technical details. On a high level, instead of spending to existing addresses, the sender uses fresh addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. These addresses were never used before, but are under the control of the respective users. With this approach, if $\mathsf{tx}^{\mathsf{er}}$ is leaked, the identities of all users on the path, especially the identity of the sender and the receiver, remain hidden. Note that we assume the input of $\mathsf{tx}^{\mathsf{er}}$ to be an unused and unlinkable input of the sender.

Fast track payments. The design considered so far has still a practical drawback compared to MHPs in the LN. In the LN, if every user is honest, the payment is carried out almost instantaneously, i.e. the channels are updated as soon as the HTLCs are opened. Obviously, users of a payment do not want to wait until some time T until the payment is carried out, even if all users are honest. To enable the same fast payments in Blitz, we extend the protocol design with an *optional* second communication round, called the fast track (we compare this second round to the one adopted in the LN below). Specifically, the users on the path can honestly update their channels from the sender to the receiver to a state where the α coins move from left to right.

For this, the sender does not go idle upon receiving the confirmation in time from the receiver. Instead, U_0 starts updating the channel $\overline{\gamma_0}$ with its neighbor U_1 to a state where the α coins are paid to U_1. Since U_0 is the only one able to publish $\mathsf{tx}^{\mathsf{er}}$, U_0 is safe when performing this update. After this update, U_1 does the same with U_2. All users on the path repeat this step until the receiver is reached. If everyone is honest, the payment will be carried out as quickly as in the LN honest case. If someone stops the update or some

honest users are skipped by colluding malicious users, honest users simply wait until time T, and claim their money (and fees) either by cooperatively updating the channel with their neighbor or forcefully on-chain. Intuitively, since intermediary users only update their right channel after updating their left channel, they cannot lose any money, even if $\mathsf{tx}^{\mathsf{er}}$ is published.

Using the fast track seems to be a better choice for normal payments. However, there are applications, where the non fast track is more suitable, e.g., a service with a trial period or a subscription model, where a user might want to set up a payment, that gets confirmed after some time. Should the user decide against it, he/she can cancel the payment. The choice of fast track is up to the user. Having this second round is completely optional and for efficiency reasons only. A payment that is carried out in one round has the same security properties as one carried out in two rounds.

Fast revoke. In the case that an intermediary is offline and the payment is unsuccessful, the refund can happen without necessarily publishing $\mathsf{tx}^{\mathsf{er}}$, saving the cost of putting a transaction on-chain. Say U_{i+1} is offline and U_i has already set up the construction with U_{i-1}. As soon as an honest U_i notices that U_{i+1} is unresponsive, it can start asking U_{i-1} to update their channel to the state before the payment was set up. After doing this, U_{i-1} asks its left neighbor to do the same and so on until the sender is reached and the payment is reverted without $\mathsf{tx}^{\mathsf{er}}$ being published. Should some intermediary refuse to honestly revoke, then $\mathsf{tx}^{\mathsf{er}}$ can still be published. Apart from funds being locked for a shorter time, one could add additional incentives to the fast revocation (or fast track) by giving a small fee to the users who are willing to participate in it. Of course, users need a mechanism to find out whether others are offline. For that, we note that the LN protocol mandates users to periodically broadcast a heartbeat message. We consider such default messages orthogonal to payment protocols and do not count them in round complexity.

Honest update. The transactions in Figure 4.4 between users are exchanged off-chain and used to guarantee that honest users do not lose any coins. However, should one of the users in a channel be able to convince the other that it is able to enforce either $\mathsf{tx}_i^{\mathsf{r}}$ or $\mathsf{tx}_i^{\mathsf{p}}$ on-chain (that is if $\mathsf{tx}^{\mathsf{er}}$ is on-chain before time T or time T has already passed, respectively), two collaborating users can simply perform an honest update. For this, they update their channel to a state where both have their corresponding balance, with the benefit that no transaction has to be put on-chain and their channel remains open.

Blitz vs. ILP/LN/AMHL. We claim that Blitz is a solution for the issues presented in Section 4.1 and allows for PCNs that have higher throughput, less communication complexity, additional security against certain attacks, and are implementable in cryptocurrencies without scripting capabilities. We highlight the differences between Blitz and other state-of-the-art payment methods such as Interledger Payments (ILP), the LN and the wormhole secure construction Anonymous Multi-Hop Locks (AMHL) [MMS+19] in Table 4.1.

First, Blitz offers balance security with only one round of communication, while ILP does not provide that and the LN requires two rounds. While the fast track optimization does

Table 4.1: Features of different payment methods: Interledger (ILP), Lightning Network (LN), Anonymous Multi-Hop Locks (AMHL), Blitz, and Blitz using the fast track payment (FT). We abbreviate timelocks as TL and signature functionality as σ. * The requirement of HTLC can be dropped from the LN using scriptless scripts when feasible.

	ILP	LN	AMHL	Blitz	Blitz FT
Bal. Security	No	Yes	Yes	Yes	Yes
Rounds	1	2	2	1	2
Atomicity	No	No (Wormhole)	Yes	Yes	Yes
Scripting	σ	σ, TL, HTLCs*	σ, TL	σ, TL	σ, TL
Collateral	n/a	linear	linear	constant	constant

Table 4.2: Collateral time for the LN, AMHL and Blitz for unsuccessful (refund) and successful payments (pay) as well as different threat models. We say *instant* when no one on the path stops the payment in either round. ξ denotes the time users need to claim their funds (e.g., in the LN 144 blocks).

	LN / AMHL		Blitz	
	refund	pay	refund	pay
anyone malicious	$n \cdot \xi$	$n \cdot \xi$	ξ	ξ
sender honest	$n \cdot \xi$	$n \cdot \xi$	Δ	ξ
everyone honest	instant	instant	instant	instant

involve a second round (from left to right, as opposed to right to left as in the LN), it is optional and affects only the efficiency (in the case everyone is honest) and not security: a payment that had a successful first round will be successful regardless of any network faults in the second round.

Indeed, the same holds true for the wormhole attack: Once a user has successfully set up a Blitz payment, it cannot be skipped anymore in the second round, even with the fast track. The payment is successful for everyone or no one, achieving thus the atomicity property missing in ILP and the LN, and honest intermediaries are not cheated out of their fees.

Secondly, Blitz reduces the collateral from linear (in the size of the path) to constant in the case some of the parties are malicious while offering comparable performance in the optimistic case, as shown in Section 4.6. For a corner case where the sender is honest, the collateral can even be unlocked almost instantaneously. We show in which cases Blitz outperforms the LN in Table 4.2. Finally, in terms of interoperability, we require only signatures and timelocks from the underlying blockchain, with the LN additionally requiring HTLCs and ILP only signatures.

Concurrent payments. In Blitz, multiple payments can be carried out in parallel, analogous to concurrent HTLC-based payments in the LN (see Appendix C.1 for further discussion and an illustrative example).

4.4 Our construction

4.4.1 Security and privacy goals

We informally review the security and privacy goals of a PCN, deferring the formal definitions to Appendix C.11.

Balance security. Honest intermediaries do not lose money [MMSK+17].

Sender/Receiver privacy. In the case of a successful payment, malicious intermediaries cannot determine if the left neighbor along the path is the actual sender or just an honest user connected to the sender through a path of non-compromised users. Similarly, malicious intermediaries cannot determine if the right neighbor is the actual receiver or an honest user connected to the receiver through a path of non-compromised users.

Path privacy. In the case of a successful payment, malicious intermediaries cannot determine which users participated in the payment aside from their direct neighbors.

4.4.2 Assumptions and building blocks

System assumptions. We assume that every party has a publicly known pair of public keys (A, B) as required for stealth address creation (see below). We further assume that honest parties are required to stay online for the duration of the protocol. Finally, we consider the route-finding algorithm an orthogonal problem and assume that every user (U_0) has access to a function $\mathsf{pathList} \leftarrow \mathsf{GenPath}(U_0, U_n)$, which generates a valid path from U_0 to U_n over some intermediaries. We refer the reader to [SVR+20, RMKG18] for more details on recent routing algorithms for PCNs. We now introduce the cryptographic building blocks that we require in our protocol.

Ledger and payment channels. We rely, as a blackbox, on a public ledger to keep track of all balances and transactions and a PCN that supports the creation, update, and closure of channels (see Section 4.2). We further assume that payment channels between users who want to conduct payments are already opened. We denote the standard operations to interact with the blockchain and the channels as follows:

$\mathsf{publishTx}(\overline{\mathsf{tx}})$: If $\overline{\mathsf{tx}}$ is a valid transaction (Section 4.2), it will be accepted on the ledger after at most time Δ.

$\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, initiates an update in $\overline{\gamma_i}$ to the state $\mathsf{tx}_i^{\mathsf{state}}$. If the update is successful, $(\mathsf{update{-}ok})$ is returned to both users of the channel, else $(\mathsf{update{-}fail})$ is returned to them. We define t_{u} as an upper bound on the time it takes for a channel update after this procedure is called.

$\mathsf{closeChannel}(\overline{\gamma_i})$: When called by a user $\in \overline{\gamma_i}.\mathsf{users}$, closes the channel, such that the latest state transaction $\mathsf{tx}_i^{\mathsf{state}}$ will appear on the ledger. We define t_{c} as an upper bound on the time it takes for $\mathsf{tx}_i^{\mathsf{state}}$ to appear on the ledger after this procedure is called.

Digital signatures. A digital signature scheme is a tuple of algorithms $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$ defined as follows:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ is a PPT algorithm that on input the security parameter λ, outputs a pair of public and private keys $(\mathsf{pk}, \mathsf{sk})$.

$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm that on input the private key sk and a message m outputs a signature σ.

$\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \sigma, m)$ is a DPT algorithm that on input the public key pk, an authentication tag σ and a message m, outputs 1 if σ is a valid authentication for m.

We require that the digital signature scheme is correct, that is, $\forall(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ it must hold that $1 \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \mathsf{Sign}(\mathsf{sk}, m), m)$. We additionally require a digital signature scheme that is strongly unforgeable against message-chosen attacks (EUF-CMA) [GMR88].

Stealth addresses [VS18]. On a high level, this scheme allows a user (say Alice) to derive a fresh public key in a digital signature scheme Σ controlled by another user (say Bob) on input two of Bob's public keys. In a bit more detail, a *stealth addresses* scheme is a tuple of algorithms $\Phi := (\mathsf{GenPk}, \mathsf{GenSk})$ defined as follows:

$(P, R) \leftarrow \mathsf{GenPk}(A, B)$ is a PPT algorithm that on input two public keys A, B controlled by some user U, creates a new public key P under U's control. This is done by first sampling some randomness $r \leftarrow^{\$} [0, l-1]$, where l is the prime order of the group used in the underlying signature scheme Σ, and computing $P := g^{\mathcal{H}(A^r)} \cdot B$, where $\mathcal{H}$ is a hash function modeled as a random oracle. Then, the value $R := g^r$ is calculated. P is the public key under U's control and R is the information required to construct the private key.

$p \leftarrow \mathsf{GenSk}(a, b, P, R)$ is a DPT algorithm that on input two secret keys a, b corresponding to the two public keys A, B and a pair (P, R) that was generated as $P \leftarrow \mathsf{GenPk}(A, B)$, creates the secret key p corresponding to P. This is done by computing $p := \mathcal{H}(R^a) + b$.

We see that correctness follows directly: $g^p = g^{\mathcal{H}(R^a)+b} = g^{\mathcal{H}(g^{r \cdot a})} \cdot g^b = g^{\mathcal{H}(A^r)} \cdot B = P$. In [VS18] it is argued that this new one-time public key P is *unlinkable* for a spectator even when observing R, meaning on a high level that P for some user U cannot be linked to any existing public key of U. For simplicity, we denote $\widetilde{U_i}, \mathsf{pk}_{\widetilde{U_i}}$ when referring to the stealth identity or the stealth public key under the control of user U_i.

Anonymous communication network (ACN). An ACN allows users to communicate anonymously with each other. One such ACN is based on onion routing, whose ideal functionality is defined in [CL05]. Sphinx [DG09] is a realization of this and (extended with a per-hop payload) is used in the Lightning Network (LN). We use this functionality here as well in a blackbox way. On a high level, routing information and a per-hop payload is encrypted and layered for every user along a path, in what is called an *onion*.

Every user on the path can then, when it is its turn, "peel off" such a layer, revealing: (i) the next neighbor; (ii) the payload meant for it; and (iii) the rest of the data, which is again an onion that can only be opened by the next neighbor. This rest of data is then forwarded to the next user and so on until the receiver is reached.

For readability, we use two algorithms, where $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\{U_i\}_{i\in[1,n]}, \{\mathsf{msg}_i\}_{i\in[1,n]})$ creates such a routing object (an onion) using (publicly known) public encryption keys of the corresponding users on the path. Moreover, when called by correct user U_i, the algorithm $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ returns $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1})$, that is, the next user on the path, a message and a new onion or returns msg_n if called by the recipient. A wrong user $U \neq U_i$ calling $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ will result in an error $\perp$.

4.4.3 2-party protocol for channel update

In this section, we show the necessary steps to update a single channel $\overline{\gamma_i}$ between two consecutive users U_i and U_{i+1} on a payment path to a state encoding our payment functionality as shown in Figure 4.4. We will describe later in Section 4.4.4 the complete multi-hop payment (MHP) protocol.

As overviewed in Section 4.3, a channel update requires to create a series of transactions to realize the "pay-or-revoke" semantics at a given channel. In particular, for readability, we define the following transaction creation methods and in Figure 4.7 some macros to be used hereby in the paper:

$\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$ This transaction takes $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$.

$\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i})$ This transaction takes as input $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. The calling user U_i makes sure that this output belongs to a stealth address under U_i's control. It creates a single output $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\alpha_i + \epsilon, \mathsf{OneSig}(U_i))$, where α_i, U_i, U_{i+1} are taken from $\mathsf{tx}_i^{\mathsf{state}}$.

We now explain in detailed order, how these transactions have to be created, signed, and exchanged. A full description in pseudocode is given in Figure 4.5. This two-party update procedure, which we call `pcSetup`, is called by a user U_i giving as parameters the channel $\overline{\gamma_i}$ with its right neighbor U_{i+1}, the transaction $\mathsf{tx}^{\mathsf{er}}$, a list containing the values R_i for the stealth addresses of each user on the path, onion_{i+1} containing some routing information for the next user, the output $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ that belongs to a stealth address of U_i, the amount to be paid α_i and the time T. The user U_i knows these values either from performing `pcSetup` with its left neighbor U_{i-1} or because U_i is the sender.

The first step for U_i is to create the new channel state from the channel $\overline{\gamma_i}$ and the amount α_i by calling $\mathsf{tx}_i^{\mathsf{state}} := \mathtt{genState}(\overline{\gamma_i}, \alpha)$. In the second step, U_i creates the transaction

$\boxed{\texttt{pcSetup}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_{\epsilon_i}, \alpha_i, T)\text{:}}$

$\boxed{U_i}$

1. $\mathsf{tx}_i^{\mathsf{state}} := \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$
2. $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_i})$
3. Send $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ to U_{i+1} $(= \overline{\gamma_i}.\mathsf{right})$

$\boxed{U_{i+1} \text{ upon } (\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}) \text{ from } U_i}$

4. Check that $\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \bot$, but returns some values $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$
5. Extract α_i and T from $\mathsf{tx}^{\mathsf{state}}$ and check $\mathsf{tx}_i^{\mathsf{state}} = \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$
6. Check that for one output $\theta_{\epsilon_x} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_x})$. If one of these previous checks failed, return $\bot$.
7. $\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$
8. Send $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$ to U_{i+1}

$\boxed{U_i \text{ upon } (\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))}$

9. If $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is not a correct signature of U_{i+1} for the $\mathsf{tx}_i^{\mathsf{r}}$ created in step 2, return $\bot$.
10. $\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$
11. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, return $\top$. Else return $\bot$.

$\boxed{U_{i+1}}$

12. Upon $(\mathsf{update{-}ok})$, return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i, T)$
13. Upon $(\mathsf{update{-}fail})$, return $\bot$

Figure 4.5: Protocol for 2-party channel update

$\mathsf{tx}_i^{\mathsf{r}}$ from $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and θ_{ϵ_i}. Then, U_i sends $\mathsf{tx}^{\mathsf{er}}$, $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, rList and onion_{i+1} to its right neighbor U_{i+1}.

Now U_{i+1} checks if $\mathsf{tx}^{\mathsf{er}}$ is well-formed and, if it is not the receiver, has an output $\theta_{\epsilon_{i+1}}$, which belongs to its stealth address (using its stealth address private keys a, b) under some $R_i \in \mathsf{rList}$. Moreover, it checks that onion_{i+1} contains the correct routing information and a message indicating that the $\mathsf{tx}^{\mathsf{er}}$ was not tampered with, for instance, a hash of it. All this is done using the macro (see Figure 4.7) $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, U_{i+1}.\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$, which returns $\bot$ if any of the checks fail.

Then, U_{i+1} checks if $\mathsf{tx}_i^{\mathsf{state}}$ and $\mathsf{tx}_i^{\mathsf{r}}$ were well-constructed and in particular, that $\mathsf{tx}_i^{\mathsf{r}}$ uses an output of $\mathsf{tx}^{\mathsf{er}}$ as input. If everything is ok, then U_{i+1} can independently create $\mathsf{tx}_i^{\mathsf{p}}$, since it requires only its own signature. Next, U_{i+1} pre-signs $\mathsf{tx}_i^{\mathsf{r}}$ and sends this signature to U_i. U_i checks if this signature is correct and then invokes a channel update with U_{i+1} to $\mathsf{tx}_i^{\mathsf{state}}$.

After this step, the `pcSetup` function is finished and returns either $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i, T)$ to U_{i+1} and $\top$ to U_i if successful or $\bot$ otherwise to the users $\overline{\gamma_i}.\mathsf{users}$. If U_{i+1} is not the receiver, it will continue this process with its own neighbor as shown in the next section.

4.4.4 Multi-hop payment description

In this section we describe the MHP protocol. The pseudocode for carrying out MHPs in Blitz is shown in Figure 4.6, the macros used in are listed in Figure 4.7. For the full description of the macros, see Appendix C.9.

Setup. Say the sender wants to pay α coins to U_n via a path $\mathsf{channelList}$ and for some timeout T. In the setup phase, the sender derives a new stealth address $\mathsf{pk}_{\widetilde{U_i}}$ and some R_i for every user except the receiver. Then, the sender creates a list rList of entries R_i and onions encoding the right neighbor U_{i+1} for every user U_i. Moreover, the sender constructs $\mathsf{tx}^{\mathsf{er}}$.

Then, it adds the sum of all per-hop fees to the initial amount α: $\alpha_i := \alpha + (n-1) \cdot \mathsf{fee}$ where fee is the fee charged by every user (see Appendix C.1). The setup ends when the sender starts the open phase with its right neighbor U_1.

Open. After successfully setting up the payment with its left user U_{i-1}, U_i knows $\mathsf{tx}^{\mathsf{er}}$, rList, onion_{i+1} α_{i-1}, T and its stealth output for $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$. Using these values and reducing α_{i-1} by fee, U_i carries out the 2-party channel update with U_{i+1}. The right neighbor continues this step with its right neighbor until the receiver is reached.

Finalize. Once the receiver has finished the open phase with its left neighbor, it sends back a signature of $\mathsf{tx}^{\mathsf{er}}$ as a confirmation to the sender, who will then check if that transaction was tampered with. If yes, or if the sender did not receive such a confirmation in time, the sender publishes $\mathsf{tx}^{\mathsf{er}}$ on the blockchain. Otherwise, the sender goes idle.

Respond. At any given time after opening a payment construction, users need to check if $\mathsf{tx}^{\mathsf{er}}$ was published. If it was, they need to refund themselves via $\mathsf{tx}^{\mathsf{r}}_i$. Also, if some user's left neighbor tries to publish $\mathsf{tx}^{\mathsf{r}}_i$ after time T, the user publishes $\mathsf{tx}^{\mathsf{p}}_i$. This ensures, that if the refund did not happen before time T, the users have a way to enforce the payment. Note that due to the relative timelock on both $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}^{\mathsf{state}}$, $\mathsf{tx}^{\mathsf{p}}_i$ will always be possible if $\mathsf{tx}^{\mathsf{er}}$ is published after T (or if the left neighbor tries to refund after T by closing the channel).

The protocol is shown in Figure 4.6. Note that we simplified the protocol for readability purposes, (e.g., by omitting the payment ids that are required for multiple concurrent payments). The full protocol modeled in the Universal Composability framework can be seen in Appendix C.10.5.

Setup

U_0 upon receiving $(\texttt{setup}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$

1. If $\texttt{checkChannels}(\mathsf{channelList}, U_0) = \bot$, abort.
2. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, abort.
3. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, abort.
4. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
5. $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
6. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$
7. $\texttt{pcSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\epsilon_0}, \alpha_0, T)$

Open

U_{i+1} upon receiving $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i, T)$

1. If U_{i+1} is the receiver U_n, send $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookrightarrow U_0$ and go idle.
2. $\texttt{pcSetup}(\overline{\gamma_{i+1}}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i - \mathsf{fee}, T)$

Finalize

U_0: Upon $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \hookleftarrow U_n$, check that $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ is U_n's valid signature for the transaction $\mathsf{tx}^{\mathsf{er}}$ created in the Setup phase. If not, or if $\mathsf{tx}^{\mathsf{er}}$ was changed, or no such confirmation was received until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{er}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{er}}))$.

Respond (Executed in every round τ_x)

1. If $\tau_x < T - t_{\mathsf{c}} - 2\Delta$ and $\mathsf{tx}^{\mathsf{er}}$ on the blockchain, $\mathsf{closeChannel}(\overline{\gamma_i})$ and, after $\mathsf{tx}_i^{\mathsf{state}}$ is accepted on the blockchain within at most t_{c} rounds, wait Δ rounds. Let $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ be a signature using the secret key $\mathsf{sk}_{\widetilde{U_i}}$. $\mathsf{publishTx}(\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})))$.
2. If $\tau_x > T$, $\overline{\gamma_i}$ is closed and $\mathsf{tx}^{\mathsf{er}}$ and $\mathsf{tx}_i^{\mathsf{state}}$ is on the blockchain, but not $\mathsf{tx}_i^{\mathsf{r}}$, $\mathsf{publishTx}(\mathsf{tx}_{i-1}^{\mathsf{p}}, (\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{p}})))$.

Figure 4.6: The Blitz payment protocol

4.5 Security analysis

4.5.1 Security model

The security model we use closely follows [AEE+21, DEF+19b, DEFM19]. We model the security of Blitz in the synchronous, global universal composability (GUC) framework [CDPW07]. We use a global ledger $\mathcal{L}$ to capture any transfer of coins. The ledger is parameterized by a signature scheme Σ and a blockchain delay Δ, which is an upper bound on the number of rounds it takes between when a transaction is posted to $\mathcal{L}$ and when said transaction is added to $\mathcal{L}$. Our security analysis is fully presented in Appendix C.10 and briefly outlined here.

Firstly, we provide an ideal functionality $\mathcal{F}_{Pay}$, which is an idealized description of the behavior we expect of our pay-or-revoke payment paradigm. This description stipulates

Macros (see Appendix C.9)

checkTxIn($\mathsf{tx}^{\mathsf{in}}$, n, U_0)**:** If $\mathsf{tx}^{\mathsf{in}}$ is well-formed and has enough coins, returns $\top$. **checkChannels**($\mathsf{channelList}, U_0$)**:** If $\mathsf{channelList}$ forms a valid path, returns the receiver U_n, else $\bot$. **checkT**(n, T)**:** If T is sufficiently large, return $\top$. Otherwise, return $\bot$ **genTxEr**(U_0, $\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}$)**:** Generates $\mathsf{tx}^{\mathsf{er}}$ from $\mathsf{tx}^{\mathsf{in}}$ along with a list of values rList to redeem their stealth addresses and an onion containing the routing information. **genState**($\alpha_i, T, \overline{\gamma_i}$)**:** Generates and returns a new channel state carrying transaction $\mathsf{tx}_i^{\mathsf{state}}$ from the given parameters, shown in Figure 4.4. **checkTxEr**($U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i$)**:** Checks if $\mathsf{tx}^{\mathsf{er}}$ is correct, U_i has a stealth address in it, and onion_i holds routing information. If unsuccessful, returns $\bot$. If U_i is the receiver, returns $(\top, \top, \top, \top, \top)$. Else, returns $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$ containing the output belonging to U_i θ_{ϵ_i}, the secret key to spend it $\mathsf{sk}_{\widetilde{U_i}}$, the next user and the next onion.

Figure 4.7: Subprocedures used in the protocol

any input/output behavior and the impact on the ledger of a payment protocol, as well as how adversaries can influence the execution. In this idealized setting, all parties communicate only with $\mathcal{F}_{Pay}$, which acts as a trusted third party.

We then provide our protocol Π formally defined in the UC framework and show that Π *emulates* $\mathcal{F}_{Pay}$. On a high level, we show that any attack that can be performed on Π can also be simulated on $\mathcal{F}_{Pay}$ or in other words that Π is at least as secure as $\mathcal{F}_{Pay}$. To prove this, we design a simulator $\mathcal{S}$, which translates any attack on the protocol into an attack on the ideal functionality. Then, we show that no PPT *environment* can distinguish between interacting with the real world and interacting with the ideal world. In the real world, the environment sends instructions to a real attacker $\mathcal{A}$ and interacts with Π. In the ideal world, the environment sends attack instructions to $\mathcal{S}$ and interacts with $\mathcal{F}_{Pay}$.

We need to show that the same messages are output in the same rounds and the same transactions are posted on the ledger in the same rounds in both the real and the ideal world, regardless of adversarial presence. To achieve this, the simulator needs to instruct the ideal functionality to output a message whenever one is output in the real protocol and the simulator needs to post the same transactions on the ledger. By achieving this, the environment cannot trivially distinguish between the real and the ideal world anymore just be looking at the messages and transactions as well as their respective timing. Formally, in Appendix C.10 we prove Theorem 4.

Theorem 4. *(informal) Let Σ be a EUF-CMA secure signature scheme. Then, for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

4.5.2 Informal security discussion

Due to space constraints, we only argue informally here why Blitz achieves security and privacy (see Section 4.4.1). We give a more formal discussion in Appendix C.11 and consider the security against some concrete attacks in Appendix C.5.

Balance security. An honest intermediary will forward a payment to its right neighbor only if first invoked by its left neighbor. If constructed correctly, the refund transactions

in both channels depend on $\mathsf{tx}^{\mathsf{er}}$ being published and the timing is identical. Also, the payment transactions have identical conditions in both channels. The only possible way for an intermediary to lose money is, if it were to pay its money to the right neighbor, while the left neighbor refunded. However, if the left neighbor is able to refund, this means that also the intermediary itself can refund. Similarly, if the right neighbor is able to claim the money, the intermediary can also claim it.

Honest sender. A sender that does not receive a confirmation from the receiver that it received the money in time, can trigger a refund by publishing $\mathsf{tx}^{\mathsf{er}}$. In the setup phase of the protocol, the sender ensures that there is enough time for this.

Honest receiver. The receiver gets the money in exchange for some service. It will wait until being certain that the money will be received before shipping the product. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof that a refund has occurred.

Privacy. Blitz requires to share with intermediaries $\mathsf{tx}^{\mathsf{er}}$, routing information and the value that is being paid. The transaction $\mathsf{tx}^{\mathsf{er}}$ uses stealth addresses for its outputs and an unlinkable input, thereby granting sender, receiver and path privacy in the honest case, as defined in Section 4.4.1. As in the LN however, the stronger notion of relationship anonymity [MMSK+17] does not hold; the payment can be linked by comparing (i) in Blitz, $\mathsf{tx}^{\mathsf{er}}$ and (ii) in the LN, the hash value. In the pessimistic case, the balance is claimed on-chain. In both Blitz and the LN, this breaks sender, path, and receiver privacy. We defer the reader to Appendix C.1 for a more detailed discussion on all privacy properties mentioned in this paragraph.

4.6 Evaluation

In this section, we evaluate the benefits that Blitz offers over the LN. The source code for our simulation is at [Bli20].

Testbed. We took a snapshot of the LN graph (October 2020) from `https://ln.bigsun.xyz/` containing 11.6k nodes, 6.5k of which have 30.9k active channels with a total capacity of 1166.7 BTC, which account for around 13.2M USD in October 2020. We ignore the nodes without active channels. The initial distribution of the channel balance is unknown. We assume that initially, the balance at each channel is available to both users. It is assigned to a user as required by payments in a first come, first serve basis. Naturally, the balance that has already been used and thus assigned to one user in the channel is not reassigned to the other user. Since we use this strategy consistently throughout all our experiments, this assignment does not introduce any bias in the results.

Simulation setup. We discretize the time in rounds and each round represents the collateral time per hop (i.e., 1 day or 144 blocks as in the LN). In such a setting, we simulate payments in batches as follows. Assume that we want to simulate *NPay* payments for an amount of *Amt* and with a failure rate of *FRate*. For that, in a first

batch, we simulate the *FRate* % of *NPay* payments, where each payment is between two nodes s and r (such that $s \neq r$) selected at random in the graph and routed through the cheapest path according to fees. Moreover, each payment in this batch is disrupted at an intermediary node chosen at random in the path between s and r. Finally, for each payment, some balance is marked to be locked at the channels for a certain number of rounds during the second batch, depending on whether we are evaluating the LN (i.e., staggered rounds) or Blitz (i.e., single round). We model thereby a setting where the network contains locked collateral due to disrupted payments.

After the first batch, we simulate a second one of *NPay* payments over 3 rounds as before, assuming that they are not disrupted (e.g., go over paths of honest nodes). We remark that here each payment may still be unsuccessful because there are not enough unlocked funds in the path between s and r. We focus thus on the effect that staggered vs. constant collateral has in the number of successful payments.

Setting parameters. Due to the off-chain nature of the LN, there is no ground truth for payment data, a common limitation in PCN-related work. We try to make reasonable assumptions for these unknown parameters in our simulation. We sample the payment amount *Amt* for each payment from the range $[1000, \mathsf{ub}]$. We use a lower bound of 1000, as technically the minimum is 546 satoshis (=1 dust) and we additionally account for fees. We select an upper bound (ub) out of $\{3000, 6000, 9000\}$, which is around 0.1%, 0.2% and 0.3% of the average channel capacity. We consider two different numbers of payments *NPay*, 78k and 978k. The former corresponds to four payments per active node and per round (ppnpr) modeling a setting with sporadic payments (e.g., a banking system), whereas the latter corresponds to 50 ppnpr, modeling a higher payment frequency (e.g., micropayments).

Finally, we vary the amount of disrupted payments *FRate* as $\{0.5, 1, 2.5\}$ % of the total payments *NPay*. We divide these disrupted payments into two groups of equal size. In the first half, the payment is stopped during the setup phase (from s to r). In the LN, the channels before the faulty/malicious node are locked with a staggered collateral lock time. In Blitz, due to the sender publishing $\mathsf{tx}^{\mathsf{er}}$, the funds are immediately unlocked. In the second half, the payment fault occurs in the second phase, which in the LN is the unlocking and in Blitz the fast track. This models the case where a node is offline or an attacker delays the completion of the payment until the last possible moment. In the LN, the collateral left of the malicious node is again staggered, whereas in Blitz the channels right of that node are locked for one simulation round. Finally, we note that distributing the disrupted payments differently into these groups will alter the results accordingly (see Appendix C.8).

Collateral effect. We calculate the number of unsuccessful payments in a baseline case (i.e., omitting the first batch of disrupted payments), in Blitz as well as in the LN and we say that $\mathsf{fail}_{\mathsf{Blitz}}$ (correspondingly $\mathsf{fail}_{\mathsf{LN}}$) is the number of payments that fail in Blitz (correspondingly the LN) when subtracting those failing also in the baseline case. We carry out every experiment for a given setting eight times and calculate the average. In

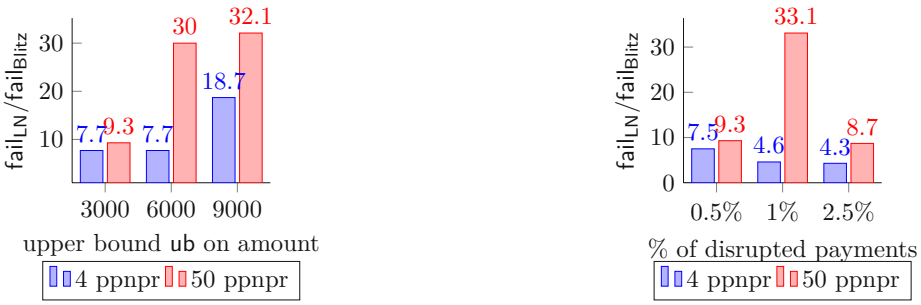

Figure 4.8: Ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. (Left) we fix the number of disrupted payments at 0.5% and vary ub. (Right) we fix ub at 3000 and vary the number of disrupted payments.

Figure 4.8 we show the ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. For all choices of parameters, there are more unsuccessful payments in the LN than in Blitz, showing thus the practical advantage of Blitz by requiring only constant collateral. We also observe that difference grows in favor of Blitz with the number of payments, showing that the advantage in terms of collateral is higher in use cases for which initially the LN was designed such as micropayments. Finally, we observe that Blitz offers higher transaction throughput even with an arguably small ratio of disrupted payments (i.e., a reduced adversarial effect).

Wormhole attack. We measure an upper bound on the amount of fees potentially at risk in the LN, due to it being prone to the wormhole attack. We observe that the amount of coins at risk grows with the number of payments and their amount. In particular, with 50 ppnpr and an upper bound of 3000 (modeling e.g., a micropayment setting), we observe that the LN put at risk 0.25 BTC (2831 USD in October 2020). Increasing the upper bound to 9000 while keeping 65 ppnpr, we observe that the LN put at risk 0.30 BTC. Blitz prevents the wormhole attack and the stealing of these fees by design.

Computation overhead. The Blitz protocol does not require any costly cryptography. In particular, it requires that each user verifies locally the signatures for the involved transactions. Moreover, each user must compute three signatures (see Figure 4.4) independently on the number of channels involved in the payment. In the LN, each user is required to compute only two signatures, one per each commitment transaction representing the new state. We remark, however, that these are all simple computations that can be executed in negligible time even with commodity hardware.

Communication overhead. We find that the contract size in Blitz is 26% smaller than the size of the HTLCs in the LN. This advantage is crucial in practice as current LN payment channels cannot hold more than 483 HTLC (and thus 483 in-flight payments) simultaneously, because otherwise, the size of the off-chain state would be higher than a valid Bitcoin transaction [TMM20, Eme]. The reduced communication overhead in Blitz implies then that it allows for more simultaneous in-flight payments per channel than in the LN.

In the pessimistic case, the LN requires to include on-chain one transaction per channel

(158 Bytes for refund, 192 Bytes for payment), while Blitz requires not only one on-chain transaction per channel (307 Bytes for refund, 158 Bytes for payment), but also that the sender includes the transaction $\mathsf{tx}^{\mathsf{er}}$ to ensure that the refund is atomic. In this sense, the LN requires a smaller overhead than Blitz for the pessimistic case. We remark that there exist incentives in PCNs for the nodes to follow the optimistic case and reduce entering the pessimistic case because it requires to close the channels and cannot be used for further off-chain payments without re-opening them, with the consequent cost in time and fees. We give detailed results about communication overhead in Appendix C.7.

4.7 Related work

PCNs have attracted plenty of attention from academia [EMSM19,KL19,TS15,MBB+19b, MMS+19] and have been deployed in practice [PD16]. These PCNs, with the exception of Interledger [TS15], follow the 2-phase-commit paradigm and suffer from (some of) the drawbacks we have discussed in this work, namely, prone to the wormhole attack, griefing attacks, staggered collateral or rely on scripting functionality not widely available. Interledger is a 1-phase protocol that however does not provide security.

Sprites [MBB+19b] is the first multi-hop payment (MHP) that achieves constant collateral. It, however, relies on Turing complete smart contracts (available in, e.g., Ethereum) thereby reducing its applicability in practice. Other constructions that require Turing complete smart contracts, e.g., State channels [DEF+19b], achieve constant collateral, but have similar privacy issues as the LN when used for MHPs. AMCU [EMSM19] achieves constant collateral and is compatible with Bitcoin. AMCU, however, reveals every participant to each other, a privacy leakage undesirable in the MHP setting.

To improve privacy, [MMSK+17] introduced MHTLCs. In [YKSN19], CHTLCs based on Chameleon hash functions were introduced, a functionality that is again not supported in most cryptocurrencies (e.g., in Bitcoin). AMHL [MMS+19] replaces the HTLC contract with novel cryptographic locks to avoid the wormhole attack. MHTLC, CHTLC, or AMHL-based MHPs all follow the 2-phase-commit paradigm and require staggered collateral. We defer to Appendix C.2 for works on 1-phase commits in the context of distributed databases.

4.8 Conclusion

Payment-channel networks (PCNs) are the most prominent solution to the scalability problem of cryptocurrencies with practical adoption (e.g., the LN). While optimistic 1-round payments (e.g., Interledger) are prone to theft by malicious intermediaries, virtually all PCNs today follow the 2-phase-commit paradigm and are thus prone to a combination of (i) security issues such as wormhole attacks; (ii) staggered collateral; and (iii) limited deployability as they rely on either HTLC or Turing complete smart contracts.

We find a redundancy implementing a 2-phase-commit protocol on top of the consensus provided by the blockchain and instead design Blitz, a multi-hop payment protocol that demonstrates for the first time that it is possible to have a 1-round payment protocol that is secure, resistant to wormhole attacks by design, has constant collateral, and builds upon digital signatures and timelock functionality from the underlying blockchain's scripting language. Our experimental evaluation shows that Blitz reduces the number of unsuccessful payments by a factor of between 4x and 33x, reduces the size of the payment contract by a 26% and saves up to 0.3 BTC (3397 USD in October 2020) in fees over a three day period as it avoids wormhole attacks by design.

Blitz can be seamlessly deployed as a (additional or alternative) payment protocol in the current LN. We believe that Blitz opens possibilities of performing more efficient and secure payments across multiple different cryptocurrencies and other applications built on top, research directions that we intend to pursue in the near future.

Acknowledgements. We thank Lloyd Fournier for his valuable feedback on an earlier version of this work. This work has been supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant P31621), the Meitner program (grant M-2608) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant 13808694) and the COMET K1 SBA; by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); by CoBloX Labs; by the National Science Foundation (NSF) under grant CNS-1846316.

CHAPTER 5

Thora: Atomic and Privacy-Preserving Multi-Channel Updates

Abstract

Most blockchain-based cryptocurrencies suffer from a heavily limited transaction throughput, which is a barrier to their growing adoption. Payment channel networks (PCNs) are one of the promising solutions to this problem. PCNs reduce the on-chain load of transactions and increase the throughput by processing many payments off-chain. In fact, any two users connected via a path of payment channels (i.e., joint addresses between the two channel end-points) can perform payments, and the underlying blockchain is used only when there is a dispute between users. Unfortunately, payments in PCNs can only be conducted securely along a path, which prevents the design of many interesting applications. Moreover, the most widely used implementation, the Lightning Network in Bitcoin, suffers from a collateral lock time linear in the path length, it is affected by security issues, and it relies on specific scripting features called Hash Timelock Contracts that hinder the applicability of the underlying protocol in other blockchains.

In this work, we present Thora, the first Bitcoin-compatible off-chain protocol that enables the atomic update of arbitrary channels (i.e., not necessarily forming a path). This enables the design of a number of new off-chain applications, such as payments across different PCNs sharing the same blockchain, secure and trustless crowdfunding, and channel rebalancing. Our construction requires no specific scripting functionalities other than digital signatures and timelocks, thereby being applicable to a wider range of blockchains. We formally define security and privacy in the Universal Composability framework and show that our cryptographic protocol is a realization thereof. In our performance

evaluation, we show that our construction requires only constant collateral, independently from the number of channels, and has only a moderate off-chain communication as well as computation overhead.

This chapter presents the results of a collaboration with Kasra Abbaszadeh and Matteo Maffei, which was published at the ACM Conference on Computer and Communications Security (CCS) in 2022 under the title "Thora: Atomic and Privacy-Preserving Multi-Channel Updates". Kasra Abbaszadeh and I contributed equally to this work and are considered to be co-first authors. Kasra Abbaszadeh and I are jointly responsible for designing the protocol and writing the paper. I am responsible for the idea, defining the security and privacy properties, as well as the application section. Kasra is mainly responsible for writing the protocol, formalization, proofs, and implementation. Matteo Maffei was the general advisor and contributed with continuous feedback.

5.1 Introduction

Permissionless cryptocurrencies such as Bitcoin [Nak09] use consensus mechanisms to verify transactions in a decentralized way and record them in a public and distributed ledger. This approach has inherent scalability issues, resulting in a low transaction throughput and a long confirmation latency. These limitations prevent cryptocurrencies from meeting the growing user demands, especially when we compare them with centralized payment networks, like Visa, which handle tens of thousands of transactions per second and confirm transactions usually within seconds.

Off-chain protocols constitute one of the most promising solutions to tackle this scalability issue. Instead of recording every transaction on the public ledger, users exchange and keep their transactions off-chain and use the ledger only as a fallback when there are disputes in order to keep their funds. One of the promising off-chain protocols is Payment Channels (PCs) which are deployed at scale in cryptocurrencies such as Bitcoin and Ethereum [PD16, MMSH16]. Intuitively, a channel is a shared address that allows two parties to maintain and update a private ledger through off-chain transactions. In a bit more detail, looking at Bitcoin's unspent transaction output (UTXO) model, users first open a PC by locking some coins in a 2-of-2 multi-signature output. Then, they can update the balance in the PC arbitrarily many times by exchanging signed transactions. Each of the users can close the PC by publishing the last state on-chain. This allows them to perform many transactions while burdening the ledger with only two transactions.

5.1.1 HTLC-based PCNs and their limitations

Payment channel networks (PCNs) such as the Lightning Network (LN) [PD16] and Raiden [Rai17] generalize this approach, by allowing two users to pay each other as long as they are connected by a path of channels with enough capacity. Such a payment in a PCN, also called a multi-hop payment (MHP), requires updating each channel on the path. The challenge here is to ensure atomicity, i.e., either all channels are updated consistently or none, such that no user is at risk of losing money. In the most popular

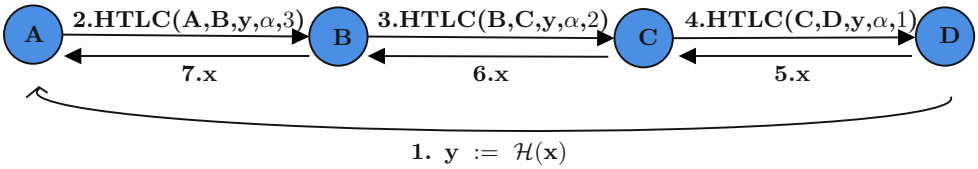

Figure 5.1: An example of a payment in LN from **A** to **D** for a value α using HTLC contracts. An HTLC contract denoted by HTLC(Alice, Bob, x, y, t), shows the following conditions: (i) If timeout t expires, Alice gets back the locked x coins. (ii) If Bob reveals a value r, such that $\mathcal{H}(r) = y$, before timeout t, Alice pays x coins to Bob.

PCN, i.e. the Lightning Network, atomicity is achieved through Hash Timelock Contracts (HTLCs) [PD16], which make the payments on each channel on the path conditioned on revealing the preimage of a certain hash. The receiver has to reveal that preimage in order to receive the money and then all intermediaries from right to left are incentivized to update their left channel in order to claim the money of the payment. An example of a payment using HTLCs is shown in Figure 5.1.

HTLC-based PCNs, however, have the following fundamental drawbacks:

Collateral. All parties on the path have to lock the payment amount α up to a period of *locktime*. The payment amount multiplied by the locktime is called *collateral*, a metric that has been used in previous work, e.g., [MBB+19b, EMSM19, AMSKM21]. In addition, parties can impose fees for the service of forwarding payments. In the case of HTLCs, each party has to lock a collateral that is linear in the size of the path n, i.e., $\Theta(\alpha \cdot n \cdot \delta)$, where δ is a security parameter defining the time by which users have to react in case of misbehavior from others (in Lightning, δ is one day).

Due to the linear collateral, the effects of *griefing attacks* [EMSM19] on HTLC-based PCNs are particularly severe. In a griefing attack, a malicious user starts a multi-hop payment to itself with the intent to block coins owned by intermediaries. The attacker manages to lock up α coins in $n-1$ honest channels. The fact that the lock duration is also linear in the path length amplifies the effects of this attack further. The malicious user subsequently lets the payment fail to limit the overall network throughput or to lock coins of specific users.

Weak atomicity. Lightning guarantees only a weak form of atomicity, that is, only the two adjacent channels of an honest node are updated consistently. In particular, Lightning is vulnerable to the *wormhole attack* [MMS+19], where two colluding malicious users can skip honest users in the phase where they reveal the preimage. This does not lead to a loss in funds for the honest users, but the malicious users can steal the fees originally intended for the honest users.

Path restriction. Since HTLC-based PCN protocols rely on an incentive-based forwarding of a preimage via a path to ensure that honest users do not lose funds,

Table 5.1: Comparing different payment methods: Lightning Network, Anonymous Multi-Hop Locks (AMHL), Sprites, Payment Trees, Atomic Multi-Channel Updates(AMCU), Blitz, and our construction. Studied features are: atomicity property, path restriction, need for Turing-complete smart contracts, size of per party collateral, and value privacy. For the latter, note that there are constructions that do not inherently leak the value transferred in individual channels, but they can only be used for applications (i.e., payments) that require the same value in all channels.

	Atomicity	Path restriction	Smart contract	pp Collateral	Value privacy
Lightning Network [PD16]	No	Yes	No	Linear	application leak
AMHL [MMS+19]	Yes	Yes	No	Linear	application leak
AMCU [EMSM19]	No	No	No	Constant	No
Payment Trees [JLT21]	Yes	Yes	No	Logarithmic	No
Blitz [AMSKM21]	Yes	Yes	No	Constant	application leak
Sprites [MBB+19b]	Yes	No	Yes	Constant	Yes
Thora	Yes	No	No	Constant	Yes

these protocols are limited to payments over a path of channels. This rules out other topologies reflecting relevant financial applications (e.g., crowd-funding can be seen as a star topology where all nodes update their channel with the beneficiary).

Value privacy. In Lightning, intermediaries implicitly learn the paid amount, as the value has to be the same (except for some fee) over all channels within the path to ensure atomicity of the protocol.

5.1.2 Related work

Recently, various protocols have been designed to overcome the aforementioned issues, but they all fall short of some properties, as summarized in Table 5.1.

Anonymous Multi-Hop Locks (AMHL) prevent the wormhole attack by dispensing from HTLCs in favor of adaptor signatures, a mechanism in which the secret is somewhat embedded in the randomness of the signature and revealed once that signature is published, but they still suffer from linear collateral and only support path-based payments.

The Atomic Multi-Channel Updates (AMCU) protocol [EMSM19] attempts to achieve payments with constant collateral and also to support more generic applications than path-formed payments. Unfortunately, AMCU is not secure: It is vulnerable to *channel closure attacks* [JLT21], where users honestly updating their channels can be the victim of double-spending attacks, which can lead to a loss of funds for honest users.

Blitz [AMSKM21] is a recently proposed payment protocol for multi-hop payments, which in contrast to Lightning requires only one round of communication through the path with constant collateral. However, Blitz supports only path-based payments.

Sprites [MBB+19b] is the only secure protocol supporting atomic multi-channel updates with constant collateral. In fact, the paper addresses only path-based payments, but we conjecture that the protocol could in principle be modified so as to support arbitrary

topologies and also to hide the paid amount. Unfortunately, Sprites inherently requires Turing-complete scripting, which makes it inapplicable to blockchain technologies with limited scripting capabilities, such as Bitcoin itself. A Turing complete scripting language provides more expressiveness, but it also enlarges the trusted computing base, opens the door to programming bugs, and makes computations more expensive (e.g., in terms of gas fees in Ethereum).

Hence, it is both a foundational and practically relevant question whether or not atomic multi-channel updates with constant collateral are possible at all in blockchains with limited scripting languages like Bitcoin. Indeed, it was conjectured in [MBB+19b] that they are not.

5.1.3 Our contribution

In this paper, we show that the aforementioned conjecture is incorrect. In particular,

- We introduce Thora, the first secure Bitcoin-compatible protocol with constant collateral for atomic, multi-channel updates. The constant collateral property not only makes the protocol financially sustainable for a large number of channels, but also mitigates the threat of griefing attacks. Thora only requires signatures and timelocks, and it is thus compatible with a number of cryptocurrencies, such as Bitcoin, Stellar, and Ripple. In addition, Thora supports payments over channels with arbitrary topologies, thereby enabling a variety of interesting applications. Finally, Thora achieves value privacy, i.e., the channel owners can synchronize their payments without necessarily disclosing the individual payment amounts.

- We formally model our protocol in the *Global Universal Composability* (GUC) framework [CDPW07], analyzing its security and privacy properties. For this, we define an ideal functionality that captures the security and privacy notions of interest and prove that Thora constitutes a GUC-realization thereof.

- We conduct a complexity analysis and performance evaluation, demonstrating the practicality of Thora.

- We instantiate Thora in the context of several applications that go beyond simple path-formed payments, such as mass payments, channel rebalancing, and crowd-funding, thereby exemplifying the class of off-chain applications enabled by Thora.

5.2 Background

In this section, we provide an overview on the background and the notations used throughout the paper. For more details, we refer the reader to [AMSKM21, AEE+21, MMSK+17].

5.2.1 UTXO based transactions

We assume the underlying blockchain to be based on the *unspent transaction output* (UTXO) model, like Bitcoin. In this model, *coins*, or the units of currency, exist in *outputs* of *transactions*. We represent each output as a tuple $\theta := (\mathsf{cash}, \phi)$ where $\theta.\mathsf{cash}$ is the output value, and $\theta.\phi$ is the condition required to spend the output. We encode the condition in the scripting language used by the underlying cryptocurrency. The notation $\mathsf{OneSig}(U)$ denotes the condition that a digital signature w.r.t. U's public key is required for spending an output. If multiple signatures are required, we write $\mathsf{MultiSig}(U_1, U_2, ..., U_n)$.

Users can transfer the ownership of outputs via transactions. A transaction spends a non-empty list of unspent outputs (transaction inputs) and maps them to a list of new unspent outputs (transaction outputs). Formally a transaction is denoted as a tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. $\mathsf{tx.id} \in \{0,1\}^*$ is the identifier, set to be the hash of inputs and outputs, $\mathsf{tx.id} = \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$, where $\mathcal{H}$ is modeled as a random oracle. $\mathsf{tx.input}$ denotes the list of identifiers of the inputs and $\mathsf{tx.output}$ denotes the list of new outputs. Also we let $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience also $\overline{\mathsf{tx}} = (\mathsf{tx}, \mathsf{witness})$ denote a full transaction. $\overline{\mathsf{tx}}.\mathsf{witness}$ consists of witnesses for the spending conditions of the transaction's inputs. Only valid transactions can be recorded on the public ledger $\mathcal{L}$ (the blockchain). A transaction is considered valid if (i) its inputs are not spent by other transactions in $\mathcal{L}$, (ii) the sum of its outputs is not greater than the sum of inputs, and (iii) the transaction provides valid witnesses fulfilling the spending conditions of every input. In practice, transactions are not recorded on the ledger and published immediately, but only after the participants in the distributed consensus accept them. We use Δ to denote the upper bound on the time it takes for a valid transaction to be published and accepted to $\mathcal{L}$.

Using the scripting language, we can encode more complex conditions on transaction outputs than simple ownerships. To better visualize transactions, we use charts in which transactions are represented as rounded rectangles and inputs as incoming arrows. Boxes inside transactions represent outputs and the values in these boxes determine the amounts of coins stored in the outputs. Outgoing arrows from an output are used to encode the condition under which said output can be spent. In particular, below an arrow, we identify who can spend an output by listing one or more public keys. A valid transaction must contain signatures that verify under these public keys. Above the arrow, we write additional conditions that are required for spending the output. These conditions can be any script supported by the scripting language of the underlying blockchain, but in this work, we only use time-locks. For denoting relative time-locks, we write $\mathsf{RelTime}(t)$ or $+t$, which means that the output can be spent only if at least t rounds have passed since the transaction holding this output was accepted on $\mathcal{L}$. For denoting absolute time-locks, we use $\mathsf{AbsTime}(t)$ or $\geq t$, which means that the output can be spent only if the round t has already passed. If an output condition is a disjunction of several conditions, i.e., $\phi = \phi_1 \vee \phi_2 \cdots \vee \phi_n$ we draw a diamond in the output box and put each condition ϕ_i below/above its own arrow. For the conjunction of several conditions,

we write $\phi = \phi_1 \wedge \phi_2 \cdots \wedge \phi_n$. We illustrate an example of our transaction charts in Figure 5.2.

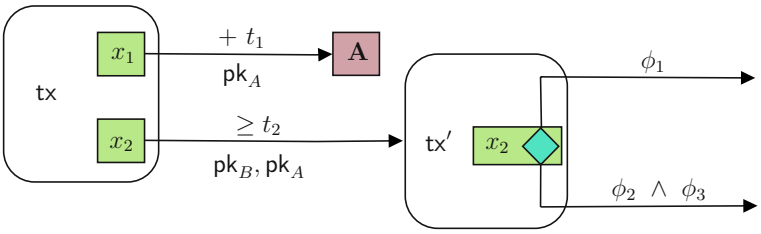

Figure 5.2: The left transaction tx has two outputs, one of value x_1 that can be spent by A, with a transaction signed w.r.t. pk_A, but only if at least t_1 rounds passed since tx is accepted on the blockchain. The other output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B at or after round t_2. The right transaction tx' has one input, which is the second output of tx containing x_2 coins, and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee (\phi_2 \wedge \phi_3)$. The inputs of tx are not shown.

5.2.2 Payment channels

Using payment channels, two users can perform an arbitrary number of payments off-chain by publishing only two transactions on the ledger, one for funding and one for closing. Through the funding transaction tx^{f}, users jointly lock up some coins in a shared multi-signature output, thereby opening a new channel. To avoid having their funds locked, the two users exchange signed transactions spending from tx^{f}, and assign new balances for users, before posting tx^{f} on-chain. Users can perform payments by exchanging new transactions that reassign their balances. These transactions holding the balances are called *states* of the channel. When the two users are done, they can close the channel by posting the last state to the ledger.

For readability, we omit the implementation details and instead use payment channels in a black-box manner, using the following abstraction: Both users have the same transaction $\mathsf{tx}^{\mathsf{state}}$, which holds the outputs representing the last state of the channel. Furthermore, we assume that the users can only publish the last $\mathsf{tx}^{\mathsf{state}}$ on the ledger. In practice, there is a punishment mechanism in place, which gives the total channel capacity to the honest party in case a malicious party publishes an old state. We refer the reader to [AEE+21, MMSK+17, MMS+19] for more details.

We denote payment channels as $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$, where $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ is the unique identifier of the channel, $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$ contains addresses of two involved parties (out of the set of all parties $\mathcal{P}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\geq 0}$ is the total number of coins in the channel and $\overline{\gamma}.\mathsf{st} := (\mathsf{output}_1, \mathsf{output}_2, ..., \mathsf{output}_n)$ is the last state of the channel and contains a list of outputs. The balance of both users can be inferred from the current state $\overline{\gamma}.\mathsf{st}$, and $\overline{\gamma}.\mathsf{balance}(P)$ returns the amount of coins owned by P for $P \in \overline{\gamma}.\mathsf{users}$. We define a

channel skeleton γ for a channel $\overline{\gamma}$, as $\gamma := (\overline{\gamma}.\mathsf{id}; \overline{\gamma}.\mathsf{users})$. Moreover, in the context of our multi-channel updates protocol, based on the direction of the payment in each channel γ, we define one of the involving parties as sender, which is denoted by $\gamma.\mathsf{sender} \in \gamma.\mathsf{users}$, and one as receiver which is denoted by $\gamma.\mathsf{receiver} \in \gamma.\mathsf{users}$.

5.2.3 Payment channel networks

A payment channel network (PCN) [MMSK+17] is a graph consisting of vertices, representing the users, and edges, representing the channels between pairs of users. PCNs enable payments between any users connected through a path of open payment channels. This is called a *multi-hop payment*. Assume user U_0 wants to pay user U_n, but there is no direct payment channel between them. Instead, U_0 has an open payment channel γ_0 with U_1, U_1 has an open payment channel γ_1 with U_2 and so on, until the receiver U_n. An MHP allows transferring coins from U_0 to U_n through intermediaries $\{U_i\}_{i\in[1,n-1]}$ atomically in a secure way, which means that no honest user is at the risk of losing money.

HTLC. The Lightning Network (LN) [PD16] achieves atomicity by using a technique called *Hash Timelock Contract* (HTLC). This contract can be executed by two parties sharing an open payment channel, e.g., Alice and Bob. First, Alice locks some of her coins in an output that is spendable if one of the following conditions is fulfilled. (i) If a specified timeout t expires, Alice gets her money back. (ii) If Bob presents a pre-image r_A for a certain hash value $\mathcal{H}(r_A)$ chosen by Alice, Bob gets the money.

An MHP in LN concatenates several HTLCs aiming for an atomic payment. In a nutshell, suppose again there is a sender U_0 who wants to pay α coins to a receiver U_n through some intermediaries $\{U_i\}_{i\in[1,n-1]}$. The payment receiver U_n chooses a random value r and sends $y = \mathcal{H}(r)$ to the sender. Then the sender sets up an HTLC with U_1 by creating a new state with three outputs $(\mathsf{output}_0, \mathsf{output}_1, \mathsf{output}_2)$ where output_0 contains α coins, output_1 contains U_0's balance minus α, and output_2 contains U_1's balance. The HTLC specifies that output_0 can be spent by U_0 if timeout $n \cdot T$ is expired, or by U_1, if she knows a value x such that $\mathcal{H}(x) = y$. Then U_1 sets up an HTLC with U_2 in a similar manner using the same hash y but a different time, $(n-1) \cdot T$. This step is repeated until the receiver is reached, with a timeout of T. We call this process the *setup phase*. Thereafter, the receiver can reveal r and claim α coins from the left neighbor. Using r, U_{n-1} can claim α coins from U_{n-2} and so on, in a second phase, which is called *open phase*. In this way, all payments can be performed atomically through the path.

Note that in the open phase, each pair of parties can either agree to update their channel to a new state off-chain, where finally U_n has α coins more, or otherwise the receiver can publish the state and a transaction with witness r on-chain. The timelocks of the HTLCs are staggered, i.e., they increase from right to left, because we need to give enough time to an intermediary party to claim her money from the left neighbor, when her right neighbor reveals r and spends the output of the corresponding HTLC. LN payments thus require (i) two rounds of pairwise, sequential communication from sender to receiver and (ii) a linear collateral lock time in terms of the path length. This opens the door to denial-of-service

attacks, also called griefing attacks [EMSM19] in the literature. Another attack that threatens the security of the HTLC-based protocols is the *wormhole* attack [MMS+19]. This attack allows two colluding users to exclude honest intermediaries from the payment and steal their fees.

Blitz. Blitz [AMSKM21] recently improved on that by requiring only one round of communication through the path, and a constant collateral lock time, while guaranteeing security in the presence of malicious intermediaries. In this protocol, the sender creates a unique transaction *Enable Refund*, which is denoted by $\mathsf{tx}^{\mathsf{er}}$. This transaction acts as a global event and makes the refunds atomic, following a *pay-unless-revoke* paradigm. On a high level, each party U_i for $i \in [0, n-1]$, creates an output of α that is spendable in two ways: (i) U_{i+1} can claim it after some specific time T, or (ii) U_i can refund the coins if $\mathsf{tx}^{\mathsf{er}}$ is on the ledger before that time T. If all channels are updated from sender to receiver in this way, the receiver sends a confirmation to the sender and the payment is considered successful. Otherwise, if any update fails, the sender posts $\mathsf{tx}^{\mathsf{er}}$ before time T to the ledger to trigger all refunds.

Note that in LN, payments in the pessimistic case are performed sequentially. In Blitz, instead, in the case of failure, all refunds can be performed in parallel whenever $\mathsf{tx}^{\mathsf{er}}$ appears on the ledger. Because of that, the collateral lock time in Blitz for each party is constant, thereby significantly reducing the effects of a griefing attack against Blitz compared to protocols with a linear collateral lock time.

5.3 Solution overview

In this work, we present Thora, the first Bitcoin-compatible protocol that enables the atomic update of arbitrary channels, going beyond the path-based topology assumed in HTLC- or Blitz-based payments. In other words, Thora supports multiple senders and receivers, without requiring them to be connected to each other. This feature enables the design of new off-chain applications as well as to perform payments across distinct PCNs sharing the same underlying blockchain. We start by informally presenting the security and privacy goals of interest and then give an intuitive overview of our construction.

5.3.1 Security and privacy goals

In this work, we focus on two fundamental properties, which we informally define below, referring the reader to Appendix D.3 for the formal definitions.

(S1) Atomicity. The aim of a multi-channel update protocol is to update a set of channels. A multi-channel update protocol achieves atomicity if there are no two channels with at least one honest user each where one update fails and the other one is successful unless at least one honest user is compensated (i.e., by getting coins she would otherwise not get). In other words, without losing coins (i) a malicious receiver cannot let the update of her channel be successful even though it should fail and (ii) a malicious sender cannot let the update fail, even though it should be successful. Note that a malicious

(irrational) user can always forfeit their own coins, e.g., by posting an old channel state, but as this is to the benefit of the honest user, we do not consider it as breaking atomicity.

(P1) Strong value privacy. We say that a multi-channel update protocol achieves value privacy if in the optimistic case (i.e., when the protocol is executed entirely offline), for each channel, no party except for the channel owners can determine the payment value. Note that this property is stronger than value privacy as defined in AMCU [MMSK+17]. In AMCU, each channel's payment value is known to all parties involved in the protocol, and the privacy of values is preserved only against parties not involved in the protocol.

Assumptions. We assume that there is a secure and authenticated channel between each protocol participant. This can be realized in practice by establishing TLS channels. Also, we do not consider the side channels that can be established by probing the nodes in the network or by observing the opening and closing on-chain operations, as these constitute orthogonal problems that affect all PCNs and can be mitigated with dedicated techniques (e.g. [DTZG22]).

5.3.2 Key idea

The approach we follow to construct our protocol is reminiscent of the *pay-unless-revoke* paradigm adopted in Blitz [AMSKM21], but it proceeds the other way around and it should thus be seen as a *revoke-unless-pay* paradigm, as discussed below. In particular, for each channel, we aim to design an update contract that simultaneously allows the receiver to claim her coins if all payments are successful and allows the sender to refund her coins if at least one channel fails to perform the payment. We propose our solution in an incremental way. First, we start with a high-level overview of the approach. Then, we discuss the challenges and possible solutions, until reaching the final protocol.

Let $\{\gamma_i\}_{i\in[1,n]}$ be the set of involved payment channels. For each channel γ_i, based on the payment direction, we define one party as the sender, denoted by $\gamma_i.\mathsf{sender}$, and one as the receiver, denoted by $\gamma_i.\mathsf{receiver}$. We call the payment value for this channel α_i. As a high-level abstraction, $\gamma_i.\mathsf{sender}$ splits α_i coins from her balance in the channel's current state and generates a new output. This output can be spent by the receiver if all payments are successful, or can be refunded to the sender if at least one payment fails. In other words, we need to overcome two challenges. First, the design should be such that *if a sender refunds her coins, then all other senders can also do that.* Second, *if the payment in a channel is successful or a receiver is able to claim her coins, then payments in all other channels are forced, and senders cannot refund.*

For the first challenge, we make all refunds possible only if a timeout T expires, so after this time, all senders can refund their coins if the coins have not been spent by the receivers. In other words, we give all users time T to finalize the payments in their channels. If the payment in a channel has not been finalized until this time, the sender can use a refund transaction and get back her coins. T is a protocol parameter, independent of the number of channels, and the same for all channels.

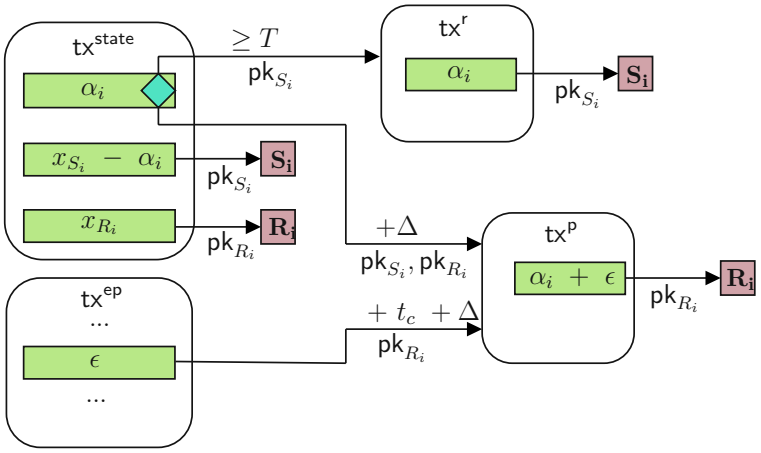

Figure 5.3: Update contract for the channel γ_i between two neighboring users γ_i.sender and γ_i.receiver with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{S_i} is the amount that $S_i = \gamma_i$.sender owns and x_{R_i} is the amount that $R_i = \gamma_i$.receiver owns in the state before $\mathsf{tx}^{\mathsf{state}}$.

For the second challenge, we make payments atomic using a global event. For each channel, the sender updates the channel and creates a payment transaction, which transfers coins to the receiver only after a global event occurs before time T. When all channels are updated correctly, senders are expected to finalize their channels, transferring coins to their receiver neighbor. In this case, if at least one receiver does not receive coins, the global event will be triggered before time T, and all payment transactions will become valid. Then, receivers can claim their cash. This global event is the appearance of a specific transaction on the ledger, which we call *Enable Payment* transaction, and denote it by $\mathsf{tx}^{\mathsf{ep}}$. This transaction is similar to *Enable Refund* transaction in the Blitz protocol, but the logic is reversed. Instead of refunds, we make payments dependent to a global event.

Update contract. For easing the presentation, let us assume first that there is a trusted user, who creates $\mathsf{tx}^{\mathsf{ep}}$ and is responsible for posting it to the ledger. $\mathsf{tx}^{\mathsf{ep}}$ contains outputs to all receivers, which is the key to achieving atomicity. We discuss the structure of the update contract below, which makes both the payment and the refund available to the channel owners. In more detail, for each channel γ_i, the sender γ_i.sender creates three transactions: $\mathsf{tx}^{\mathsf{state}}$, tx^{r}, and tx^{p}. $\mathsf{tx}^{\mathsf{state}}$ is a new state transaction, where α_i coins from the sender are put in a contract that can be spent by the other two transactions. Transaction tx^{r} refunds back the α_i coins to the sender if a timeout T expires. Transaction tx^{p} has inputs from $\mathsf{tx}^{\mathsf{ep}}$ and $\mathsf{tx}^{\mathsf{state}}$ and transfers the coins to the receiver if $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger before time T. The design of these transactions is shown in Figure 5.3. The sender sends $\mathsf{tx}^{\mathsf{state}}$ and the signed tx^{p} to the receiver, who verifies the messages and updates the channel to the new state $\mathsf{tx}^{\mathsf{state}}$ together with the sender. In the case of success, the receiver sends an endorsement to the trusted user.

Atomic payments. If the trusted user receives endorsements from all receivers, she informs all parties to finalize their channels and to transfer coins to receivers safely. There are two error cases. (i) The trusted user does not receive the endorsement from every receiver. In this case, no party will get a message from the trusted user to finalize the channel, so all channels are safe, and after time T they can be restored to the initial state based on refund transactions. (ii) If a sender gets the *finalize* message from the trusted user but does not finalize her channel, the corresponding receiver informs the trusted user to put $\mathsf{tx}^{\mathsf{ep}}$ on-chain before time T in order to force all payments.

At this point, our goal is to eliminate the trusted user assumption. Indeed, if we elected one of the parties for creating and publishing $\mathsf{tx}^{\mathsf{ep}}$, that party might act maliciously and break atomicity. For instance, by not posting $\mathsf{tx}^{\mathsf{ep}}$ to the ledger when some senders do not finalize their channel, or by posting $\mathsf{tx}^{\mathsf{ep}}$ when some channels have been updated with $\mathsf{tx}^{\mathsf{state}}$ and some not, payments would no longer be atomic. Our strategy is thus to enable all receivers to publish $\mathsf{tx}^{\mathsf{ep}}$, but only after every channel updated already to $\mathsf{tx}^{\mathsf{state}}$. For this, each receiver creates her own $\mathsf{tx}^{\mathsf{ep}}$. Each $\mathsf{tx}^{\mathsf{ep}}$ has an input conditioned on the public keys of the creator and of all senders, and it has outputs to all receivers. An example of this transaction is shown in Figure 5.4.

All receivers send their $\mathsf{tx}^{\mathsf{ep}}$ to all other parties, and this time each sender creates one tx^{p} per $\mathsf{tx}^{\mathsf{ep}}$. Then, for each channel, the sender and the receiver jointly update the channel using $\mathsf{tx}^{\mathsf{state}}$ as we discussed earlier. If no error occurs, the receiver sends a first endorsement to all parties instead of the trusted user. Each sender waits until receiving all endorsements to make sure that all channels are updated using $\mathsf{tx}^{\mathsf{state}}$. After that, the sender sends her signature to each $\mathsf{tx}^{\mathsf{ep}}$ to the creator. Eventually, when all receivers get complete signatures to their $\mathsf{tx}^{\mathsf{ep}}$, they send their second endorsement and the senders are safe to start finalizing channels and transfer coins to the receivers because all channels have been updated with $\mathsf{tx}^{\mathsf{state}}$. If some transfer fails, the receivers can post $\mathsf{tx}^{\mathsf{ep}}$ on the ledger and force all payments.

We now intuitively argue why atomicity and strong privacy hold. For atomicity, an honest sender will only update the channel with her receiver neighbor, if she receives the second endorsement from all receivers, which means that every receiver is able to force payments via $\mathsf{tx}^{\mathsf{ep}}$. Similarly, honest receivers will only give their second endorsement if they received all the signatures from $\mathsf{tx}^{\mathsf{ep}}$. This means that if a malicious user does not send her signature or endorsement to any or some of the users, this will not break atomicity but potentially only prevent updates from taking place or force the updates via some $\mathsf{tx}^{\mathsf{ep}}$. Moreover, if a malicious receiver sends either endorsement prematurely, she will only potentially lose money without side effects to other channels, i.e., the adversary will donate money to the sender without affecting the payments in the other channels. Finally, malicious users are rational, which means they will either refund their money or claim the money from a forced update, if possible.

With regards to privacy, the payment value is only known to the sender and the receiver, and in particular, it is not disclosed to the other parties involved in the protocol.

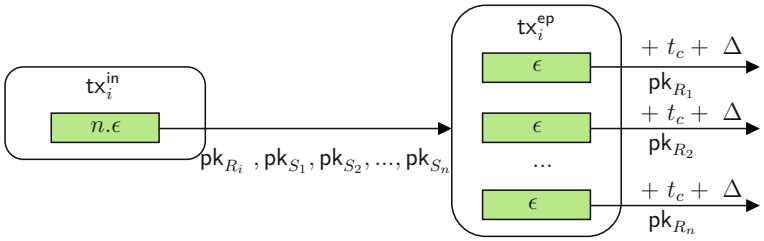

Figure 5.4: Transaction $\mathsf{tx}_i^{\mathsf{ep}}$ created by receiver R_i for a payment with n channels, where the set of all senders is $\{S_j\}_{j\in[1,n]}$ and the set of all receivers is $\{R_j\}_{j\in[1,n]}$. This transaction enables all payments and spends the output of transaction $\mathsf{tx}_i^{\mathsf{in}}$.

Timelocks. tx^{p} should be valid until time T, and tx^{r} should be valid after that time. The latter can easily be handled by using an absolute timelock of T, which is supported by the underlying scripting language of most cryptocurrencies, including Bitcoin. However, we do not have access to scripting functionalities to define outputs that are valid before time T.

We can solve this problem by applying relative timelocks. In particular, we add a relative timelock of Δ for the transaction tx^{p}, where Δ is the blockchain delay. According to this timelock, if $\mathsf{tx}^{\mathsf{state}}$ appears on the ledger after time T, users have enough time to post tx^{r} before the relative timelock of tx^{p} expires. In other words, tx^{r} is always accepted over tx^{p}, in the case that both are published after time T. On the other hand, if $\mathsf{tx}^{\mathsf{state}}$ appears before time $T - \Delta$, users have enough time to post tx^{p} and force the payment.

One other issue we should consider is the unfair advantage of a receiver who closes her channel in advance and puts her $\mathsf{tx}^{\mathsf{ep}}$ on the ledger just before time $T - \Delta$. In this case, the receiver can post tx^{p} and force the payment in her channel, but other receivers, who have not closed their channels, do not have enough time to react to $\mathsf{tx}^{\mathsf{ep}}$. To prevent this issue and give enough time to all users to close their channels and post tx^{p} to the ledger, we add a relative time of $t_c + \Delta$ to the outputs of $\mathsf{tx}^{\mathsf{ep}}$, where t_c is an upper bound on the time a user needs to close a channel (Figure 5.3). For more detail on how we prevent race conditions, we refer the reader to Section 5.8.

We point out that, as in the Lightning Network, honest users are assumed to be online and to monitor the ledger. This assumption is orthogonal to our construction and can be removed using the techniques proposed in the literature for this purpose, e.g., Watchtowers [MBB+19a, ATLW20].

Protocol overview. To wrap up, our protocol proceeds in four main phases, as described below and visualized in Figure 5.5.

1. **Pre-Setup**: Each receiver creates her own $\mathsf{tx}^{\mathsf{ep}}$, and sends it to all other parties. Each $\mathsf{tx}^{\mathsf{ep}}$, in addition to the creator's signature, requires signatures from all senders and has one output for each receiver.

2. **Setup**: The senders create $\mathsf{tx}^{\mathsf{state}}$ and tx^{r}, and also one tx^{p} per $\mathsf{tx}^{\mathsf{ep}}$. They send $\mathsf{tx}^{\mathsf{state}}$ and all tx^{p} to their receiver neighbor. Also, they include their signatures for every tx^{p} in the message to their receiver neighbor. This ensures that receivers can post tx^{p} on the ledger regardless of which $\mathsf{tx}^{\mathsf{ep}}$ is posted in the end. Eventually, the receivers verify the messages and send their first endorsement to all parties.

3. **Confirmation**: When a sender gets all such endorsements, she is sure that all channels have been updated by $\mathsf{tx}^{\mathsf{state}}$. Then, the sender signs each $\mathsf{tx}^{\mathsf{ep}}$ and sends it to the corresponding receiver. When a receiver gets the signatures from all senders, she is able to post her $\mathsf{tx}^{\mathsf{ep}}$ on the ledger, so she sends a second endorsement to all parties.

4. **Finalizing**: When the senders get the second endorsement from all receivers, they know that all receivers are able to put their $\mathsf{tx}^{\mathsf{ep}}$ on the ledger, so they can start updating their channels safely. When one update fails and the corresponding receiver does not get the coins, she checks if a $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger or else posts her own $\mathsf{tx}^{\mathsf{ep}}$. Either way, she claims her coins via some tx^{p}.

Fast payments. Similar to the Lightning Network, in the case that all users are honest, updates can be carried out almost instantaneously, i.e., the channels are updated as soon as the second endorsements are received from receivers. When the senders are ensured that each receiver has all signatures required for spending her $\mathsf{tx}^{\mathsf{ep}}$, they can safely update their channels and pay coins to their right neighbors.

Honest update. The update contract and the corresponding transactions $\mathsf{tx}^{\mathsf{state}}$, tx^{r}, and tx^{p} are exchanged between two parties sharing a channel to guarantee that honest users do not lose their coins and atomicity holds during the protocol execution. However, when one of the two-channel owners is able to convince the other one that she is able to force the payment (or refund) by posting tx^{p} (or tx^{r}) to the ledger, the two parties can update the channel honestly to a state on which both agree. In other words, when both parties of a channel are honest, no on-chain transaction is required.

5.4 Construction

5.4.1 Building blocks

Digital signatures. A digital signature scheme consists of three algorithms: KeyGen, Sign, Vrfy.
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ is a PPT algorithm, taking the security parameter 1^λ as input and returning a public key pk and the corresponding secret key sk.
$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm, taking a secret key sk and a message m as inputs and returning a signature σ.
$\{0,1\} \leftarrow \mathsf{Vrfy}(\sigma, m, \mathsf{pk})$ is a DPT algorithm, taking signature σ, a message m, and a

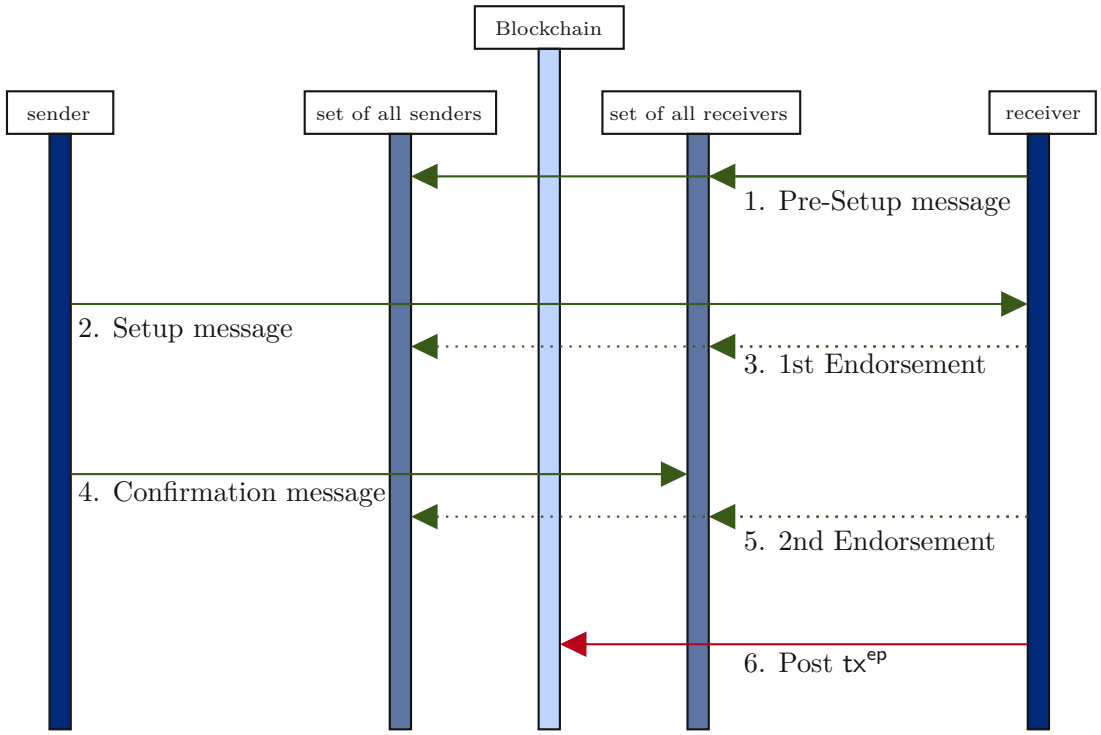

Figure 5.5: For each channel, first, the receiver sends her own $\mathsf{tx}^{\mathsf{ep}}$ to all other parties (the Pre-Setup message). The sender creates $\mathsf{tx}^{\mathsf{state}}$ and one tx^{p} for each $\mathsf{tx}^{\mathsf{ep}}$, then sends all these transactions to the receiver (Setup message). After verifying the message, the receiver sends her first endorsement to all other parties. When the sender gets all endorsements, she sends her signature to each $\mathsf{tx}^{\mathsf{ep}}$ to its creator (Confirmation message). After getting all signatures and verifying them, the receiver sends the second endorsement to all other parties. Finally, when the receiver has enough signatures as her $\mathsf{tx}^{\mathsf{ep}}$ witnesses, and the payment is not received, she will post her $\mathsf{tx}^{\mathsf{ep}}$ to the ledger.

public key pk as inputs, and returning 1 if σ is a valid signature on message m and created by the secret key corresponding to pk. Otherwise, it returns 0.

Ledger and payment channels. In this work, we use a ledger and a PCN as black boxes. The ledger keeps a record of the balances of users and all transactions. The PCN supports the operations *open*, *close*, and *update*. For simplicity, we assume the payment channels involved in the multi-channel updates protocol to be already open. We assume that ledger and PCN expose the following API to the users:

- getBalance(U): Returns the sum of all coins in the UTXOs owned by user U on the ledger.
- splitCoins(U, v, ϕ): Aggregates all UTXOs owned by U and returns a transaction with

an output containing v coins, which is conditioned on ϕ. If the balance of U is greater than v, the rest is sent to an address controlled by U. If the balance of U is less than v, the procedure returns $\perp$.

- publishTx($\overline{\mathsf{tx}}$): Appends the transaction $\overline{\mathsf{tx}}$ to the ledger after at most Δ rounds, if witnesses are valid, inputs exist and are unspent, and the sum of coins in their outputs is less than or equal to the sum of coins in the inputs.

- updateChannel($\overline{\gamma}, \mathsf{tx}^{\mathsf{state}}$): Initiates an update in the channel $\overline{\gamma}$ to the state defined by $\mathsf{tx}^{\mathsf{state}}$, when called by a user $\in \overline{\gamma}.\mathsf{users}$. The update is performed after at most t_u rounds. Upon the termination, the procedure returns `UPDATE-OK` in the case of success, and `UPDATE-FAIL` in the case of failure to both users.

- closeChannel($\overline{\gamma}$): Closes the channel $\overline{\gamma}$ when called by a user $\in \overline{\gamma}.\mathsf{users}$. The latest state $\overline{\gamma}.\mathsf{st}$ appears on the ledger after at most t_c rounds.

5.4.2 Protocol description

Let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of all updates, where $\{\gamma_i\}_{i\in[1,n]}$ denotes the involved payment channels and α_i denotes the payments value through the channel γ_i. Let dealer be the trigger party, $\mathcal{S} := \{\gamma_i.\mathsf{sender}\}_{i\in[1,n]}$ and $\mathcal{R} := \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$ the set of all senders and all receivers respectively. $\mathcal{S}$ and $\mathcal{R}$ are known to all parties. A simplified version of the Thora protocol and the used macros are shown below. We refer the reader to Appendix D.2.5 for a full description of the protocol. The main phases of the protocol are as follows.

Initialization. First, we make sure that all parties are aware of every channel that is participating in the update. The protocol then starts from the *Pre-Setup* phase. The protocol execution is triggered by a party denoted by dealer. Note that the triggering party has no security or privacy advantages over the others.

Pre-Setup. Each user $\gamma_i.\mathsf{receiver}$ creates $\mathsf{tx}_i^{\mathsf{in}}$, which has an output conditioned on the public keys of $\gamma_i.\mathsf{receiver}$ and all senders in $\mathcal{S}$. The value of the output is $n \cdot \varepsilon$, where ε is the smallest possible amount of cash. $\mathsf{tx}_i^{\mathsf{in}}$ is created by calling the procedure `GenTxIn`. Then, $\gamma_i.\mathsf{receiver}$ calls `GenTxEp`, which takes $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathcal{R}$ as inputs, and returns a transaction $\mathsf{tx}_i^{\mathsf{ep}}$ with outputs to all users in $\mathcal{R}$, each containing ε coins. $\gamma_i.\mathsf{receiver}$ sends $\mathsf{tx}_i^{\mathsf{ep}}$ to all users. The structure of $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathsf{tx}_i^{\mathsf{ep}}$ can be viewed in Figure 5.4.

Setup. $\gamma_i.\mathsf{sender}$, upon receiving $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$ from all receivers, verifies the correctness of these transactions. Then, $\gamma_i.\mathsf{sender}$ creates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. $\mathsf{tx}_i^{\mathsf{state}}$ splits α_i coins from the sender's current balance in $\gamma.\mathsf{st}$, which is spendable by payment or refund transactions. $\mathsf{tx}_i^{\mathsf{r}}$ returns the coins back to $\gamma_i.\mathsf{sender}$ only if the time T elapses. $\mathsf{tx}_{i,j}^{\mathsf{p}}$ has an input from $\mathsf{tx}_j^{\mathsf{ep}}$ and sends the split coins to $\gamma_i.\mathsf{receiver}$. The sender creates $\mathsf{tx}_i^{\mathsf{state}}$ by the procedure `GenState`, $\mathsf{tx}_i^{\mathsf{r}}$ by the procedure `GenRef`, and $\mathsf{tx}_{i,j}^{\mathsf{p}}$ by the procedure `GenPay`. $\gamma_i.\mathsf{sender}$ sends $\mathsf{tx}_i^{\mathsf{state}}$ and all signed $\mathsf{tx}_{i,j}^{\mathsf{p}}$ to the receiver neighbor. We refer

the reader to Figure 5.3 for the structure of these transactions. γ_i.receiver checks the correctness of the transactions and signatures, then sends the first endorsement to all parties.

Confirmation. When a sender γ_i.sender gets first endorsements from all parties in $\mathcal{R}$, it updates γ_i using $\mathsf{tx}_i^{\mathsf{state}}$. If the update is performed successfully, γ_i.sender sends a signature on each $\mathsf{tx}_j^{\mathsf{ep}}$ to the receiver γ_j.receiver. Each receiver γ_i.receiver waits for all signatures on $\mathsf{tx}_i^{\mathsf{ep}}$ and then sends the second endorsement to all parties if γ_i has been updated successfully.

Finalizing. Upon receiving the second endorsement from all parties in $\mathcal{R}$, a sender can safely update the channel to its final state with the receiver neighbor. When updating a channel fails in this phase, and no $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger, the receiver can post her $\mathsf{tx}^{\mathsf{ep}}$ and force the payment.

Respond. This phase is executed in every round by all users. Each sender γ_i.sender checks whether the current round is greater than T, γ_i has been closed, and at least one $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger. If so, γ_i.sender posts $\mathsf{tx}_i^{\mathsf{r}}$ to the ledger before γ_i.receiver force the payment by posting a payment transaction. On the other side, each receiver γ_i.receiver checks whether one $\mathsf{tx}_j^{\mathsf{ep}}$ has appeared on the ledger. If so, she closes the channel γ_i. After the appearance of $\mathsf{tx}_i^{\mathsf{state}}$ on the ledger, she posts $\mathsf{tx}_{i,j}^{\mathsf{p}}$ to the ledger and forces the payment through the channel γ_i.

The Thora multi-channel updates protocol

- Let dealer be a selected user as the trigger party, T the upper bound on the time we expect the updates to be performed, and Δ the blockchain delay.
- Let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of all ongoing updates. Each α_i is known only for parties in γ_i.users.

Initialization

dealer

1. Send message $(\texttt{init}, \{\gamma_i\}_{i\in[1,n]})$ to all parties in $\{\gamma_i.\mathsf{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$.

All parties upon receiving $(\texttt{init}, \{\gamma_i\}_{i\in[1,n]})$ from dealer

1. Verify the channels set. If the decision is not participating in the protocol, return `abort`.
2. Set $\mathcal{S} := \{\gamma_i.\mathsf{sender}\}_{i\in[1,n]}$, $\mathcal{R} := \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$, and $\mathcal{P} := \mathcal{S} \cup \mathcal{R}$.
3. Go to the *Pre-Setup* phase.

Pre-Setup

γ_i.receiver

1. Set $\mathsf{tx}_i^{\mathsf{in}} := \mathtt{GenTxIn}(\gamma_i.\mathsf{receiver}, \{\gamma_k\}_{k\in[1,n]})$.
2. Set $\mathsf{tx}_i^{\mathsf{ep}} := \mathtt{GenTxEp}(\{\gamma_k\}_{k\in[1,n]}, \mathsf{tx}_i^{\mathsf{in}})$.
3. Send $\mathsf{tx}_i^{\mathsf{ep}}$ to all parties in $\mathcal{R} \cup \mathcal{S}$.

All users upon receiving $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$ from all parties in $\mathcal{R}$

1. For all $j \in [1,n]$, If $\mathtt{CheckTxEp}(\mathsf{tx}_j^{\mathsf{ep}}, \gamma_j.\mathsf{receiver}, \{\gamma_k\}_{k\in[1,n]}) = \perp$, return `abort`.
2. Go to the *Setup* phase.

Setup

γ_i.sender

1. Set $\mathsf{tx}_i^{\mathsf{state}} = \mathtt{GenState}(\alpha_i, T, \overline{\gamma}_i)$.
2. Set $\mathsf{tx}_i^{\mathsf{r}} = \mathtt{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{sender})$.
3. For all $j \in [1,n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to γ_i.receiver, then create $\mathsf{tx}_{i,j}^{\mathsf{p}} := \mathtt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$ and the corresponding signature $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})$.
4. Send $(\mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$ to γ_i.receiver.

γ_i.receiver upon receiving

$(\mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$ from γ_i.sender

1. If $\mathsf{tx}_i^{\mathsf{state}} \neq \mathtt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, return `abort`.
2. If any signature $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})$ is not correct, return `abort`.
3. For all $j \in [1,n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ owned by γ_i.receiver. if $\mathsf{tx}_{i,j}^{\mathsf{p}} \neq \mathtt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$, return `abort`.
4. Send message $(\mathtt{setup\text{-}ok}_i)$ to all parties in $\mathcal{P}$.

All users upon receiving $\{(\mathtt{setup\text{-}ok}_j\}_{j\in[1,n])}$

from all parties in $\mathcal{R}$

1. Go to the *Confirmation* phase.

Confirmation

γ_i.sender

1. updateChannel$(\overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}})$.
2. If time t_u has expired and the message (UPDATE-OK) has not been returned, return abort.
3. For all $j \in [1, n]$, send $\sigma(\mathsf{tx}_j^{\mathsf{ep}})$ to γ_j.receiver.

γ_i.receiver upon receiving $\{\sigma(\mathsf{tx}_i^{\mathsf{ep}})\}_{j\in[1,n]}$) from all parties in $\mathcal{S}$

1. If (UPDATE-OK) has been returned and for all $j \in [1, n]$, $\sigma(\mathsf{tx}_j^{\mathsf{ep}})$ is a valid signatures, send message (confirmation-ok$_i$) to all parties in $\mathcal{P}$, otherwise return abort.

All users upon receiving { (confirmation-ok$_j$ }$_{j\in[1,n])}$

from all parties in $\mathcal{R}$

1. Go to the *Finalizing* phase.

Finalizing

γ_i.sender

1. Set $\mathsf{tx}_i^{\mathsf{trans}}$ = GenTrans$(\alpha_i, \overline{\gamma}_i)$.
2. updateChannel$(\overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{trans}})$.

γ_i.receiver

1. If the message (UPDATE-OK) has not been received for the final transfer, and no $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger, before time $T - t_c - 3\Delta$, combine received signatures from senders for $\mathsf{tx}_i^{\mathsf{ep}}$ with own signature inside $\sigma(\mathsf{tx}_i^{\mathsf{ep}})$ and calls publishTx$(\mathsf{tx}_i^{\mathsf{ep}}, \sigma(\mathsf{tx}_i^{\mathsf{ep}}))$.

Respond(Executed in every round τ_x)

γ_i.receiver

1. If $\tau_x < T - t_c - 2\Delta$ and at least one $\mathsf{tx}^{\mathsf{ep}}$ is on-chain, closeChannel($\overline{\gamma}_i$).
2. After $\mathsf{tx}_i^{\mathsf{state}}$ is accepted on the blockchain within at most t_c rounds, wait Δ rounds. Let $\sigma(\mathsf{tx}_i^{\mathsf{p}})$ be a signature using the secret key $sk_{\gamma_i.\mathsf{receiver}}$ in addition to received signature from $\gamma_i.\mathsf{sender}$ for $\mathsf{tx}_i^{\mathsf{p}}$. publishTx($\mathsf{tx}_i^{\mathsf{p}}, \sigma(\mathsf{tx}_i^{\mathsf{p}})$).

$\gamma_i.\mathsf{sender}$

1. If $\tau_x > T$, $\overline{\gamma}_i$ is closed and $\mathsf{tx}_i^{\mathsf{state}}$ and at least one $\mathsf{tx}^{\mathsf{ep}}$ is on the ledger, but not $\mathsf{tx}_i^{\mathsf{p}}$, publishTx($\mathsf{tx}_i^{\mathsf{r}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{r}})$).

Subprocedures used in the multi-channel updates protocol

GenTxIn(R, $\{\gamma_k\}_{k\in[1,n]}$):

1. $n := |\{\gamma_k\}_{k\in[1,n]}|$
2. $\phi := \mathsf{MultiSig}(R, \gamma_1.\mathsf{sender}, \gamma_2.\mathsf{sender}, ..., \gamma_n.\mathsf{sender})$.
3. Return $\mathsf{tx}^{\mathsf{in}} := \mathrm{splitCoins}(R, n \cdot \varepsilon, \phi)$.

GenTxEp($\{\gamma_k\}_{k\in[1,n]}$, $\mathsf{tx}^{\mathsf{in}}$):

1. $n := |\{\gamma_k\}_{k\in[1,n]}|$
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \leq n \cdot \varepsilon$, return $\perp$.
3. $\mathsf{outputList} := \emptyset$.
4. For each $R_i := \gamma_i.\mathsf{receiver}$ for all $i \in [1, n]$:
   - $\mathsf{outputList} = \mathsf{outputList} \cup (\varepsilon, \mathsf{OneSig}(R_i) \wedge \mathsf{RelTime}(t_c + \Delta))$
5. $id := \mathcal{H}(\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$.
6. Return $\mathsf{tx}^{\mathsf{ep}} := (id, \mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$.

CheckTxEp($\mathsf{tx}^{\mathsf{ep}}$, R,$\{\gamma_k\}_{k\in[1,n]}$):

1. $n := |\{\gamma_k\}_{k\in[1,n]}|$
2. If $\mathsf{tx}^{\mathsf{ep}}.\mathsf{input}.\mathsf{cash} \leq n \cdot \varepsilon$ or $\mathsf{tx}^{\mathsf{ep}}.\mathsf{input}.\phi \neq \mathsf{MultiSig}(R, \gamma_1.\mathsf{sender}, \gamma_2.\mathsf{sender}, ..., \gamma_n.\mathsf{sender})$, return $\perp$.
3. If $|\mathsf{tx}^{\mathsf{ep}}.\mathsf{output}| \neq n$, return $\perp$.

4. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{ep}}.\mathsf{output}$ if $\mathsf{cash} \neq \varepsilon$ or $\phi \neq (\mathsf{OneSig}(x), \mathsf{RelTime}(t_c + \Delta))$, where x is one of the receivers, return $\perp$.
5. Return $\top$.

GenState$(\alpha, T, \overline{\gamma})$**:**

1. Let $\theta' := \overline{\gamma}.\mathsf{st}$ be the current state of channel $\overline{\gamma}$ and contains two outputs $\theta'_s = (x_s, \mathsf{OneSig}(\overline{\gamma}.\mathsf{sender}))$ and $\theta'_r = (x_r, \mathsf{OneSig}(\overline{\gamma}.\mathsf{receiver}))$.
2. If $x_s < \alpha$ return $\perp$.
3. Return $\theta := (\theta_0, \theta_1, \theta_2)$ such that:
   - $\theta_0 := (\alpha, (\mathsf{OneSig}(\overline{\gamma}.\mathsf{sender}) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(\overline{\gamma}.\mathsf{sender}, \overline{\gamma}.\mathsf{receiver}) \wedge \mathsf{RelTime}(t_c + \Delta)))$
   - $\theta_1 := (x_s - \alpha, \mathsf{OneSig}(\overline{\gamma}.\mathsf{sender}))$
   - $\theta_2 := (x_r, \mathsf{OneSig}(\overline{\gamma}.\mathsf{receiver}))$

GenRef$(\mathsf{tx}^{\mathsf{state}}, \gamma_i.\mathsf{sender})$**:**

1. Return a transaction tx^{r} such that $\mathsf{tx}^{\mathsf{r}}.\mathsf{input} := \mathsf{tx}^{\mathsf{state}}.\mathsf{output}[0]$ and $\mathsf{tx}^{\mathsf{r}}.\mathsf{output} := (\mathsf{tx}^{\mathsf{state}}.\mathsf{output}[0].\mathsf{cash}, \mathsf{OneSig}(\gamma_i.\mathsf{sender}))$.

GenPay$(\mathsf{tx}^{\mathsf{state}}, \gamma.\mathsf{receiver}, \theta)$**:**

1. Return a transaction tx^{p} such that $\mathsf{tx}^{\mathsf{p}}.\mathsf{input} := (\mathsf{tx}^{\mathsf{state}}.\mathsf{output}[0], \theta)$ and $\mathsf{tx}^{\mathsf{p}}.\mathsf{output} := (\mathsf{tx}^{\mathsf{state}}.\mathsf{output}[0].\mathsf{cash} + \theta.\mathsf{cash}, \mathsf{OneSig}(\gamma.\mathsf{receiver}))$.

GenTrans$(\alpha, \overline{\gamma})$**:**

1. Let $\theta' := \overline{\gamma}.\mathsf{st} = (\theta'_0, \theta'_1, \theta'_2)$ be the current state of channel $\overline{\gamma}$.
2. Return $\theta := (\theta_0, \theta_1)$ such that:
   - $\theta_0 := (\theta'_1.\mathsf{cash}, \mathsf{OneSig}(\overline{\gamma}.\mathsf{sender}))$
   - $\theta_1 := (\theta'_2.\mathsf{cash} + \alpha, \mathsf{OneSig}(\overline{\gamma}.\mathsf{receiver}))$

5.5 Security analysis

5.5.1 Security model

We model the security of our multi-channel updates protocol in the synchronous setting and global universal composability (GUC) framework [CDPW07]. Our security model is

similar to the one adopted in prior work [AEE+21, AMSKM21, DEF+19b]. In particular, the global ledger $\mathcal{L}$ is modeled by the functionality $\mathcal{G}_{Ledger}$, which is parameterized by a signature scheme Σ and a blockchain delay Δ. We model the notion of communication by the ideal functionality $\mathcal{F}_{GDC}$ and the time by $\mathcal{G}_{clock}$. Moreover, we define an ideal functionality $\mathcal{F}_{Channel}$, which provides *open*, *update*, and *close* operations for payment channels.

The formal security analysis is detailed in Appendix D.2. In this section, we briefly present a high-level overview of the security model. First, we provide an ideal functionality $\mathcal{F}_{update}$, which describes an ideal multi-channel update protocol with atomicity and strong value privacy properties. $\mathcal{F}_{update}$ is parameterized by a blockchain delay Δ and a time T, which determine an upper bound on the expected time for a successful Thora payment. The ideal functionality describes the input/output behaviors of the payment protocol users and their impacts on the global ledger.

We then describe the Thora protocol Π formally and show that Π GUC-realizes $\mathcal{F}_{update}$. Intuitively, this means that we design a simulator $\mathcal{S}$, which translates any attack on the protocol Π on the ideal functionality $\mathcal{F}_{update}$. We then show that no PPT environment can distinguish between interacting with the real world and interacting with the ideal world. Thus, Π provides both atomicity and strong value privacy. This is stated by Theorem 5 and formally proven in Appendix D.2.

Theorem 5. *For any $\Delta, T \in \mathbb{N}$, the protocol Π GUC-realizes the ideal functionality $\mathcal{F}_{update}$.*

5.5.2 High-level functionality description

We give a high-level description of our channel update ideal functionality $\mathcal{F}_{update}$ and refer to Appendix D.2 for the formal UC description. $\mathcal{F}_{update}$ can be called for a set of channels to be updated, essentially with the goal of atomically performing payments in each channel from sender to receiver. Similar to the protocol, the ideal functionality proceeds in the following phases.

In the *initialization* phase, the set of channels to be updated is registered with $\mathcal{F}_{update}$. This phase is initiated by a dealer, which can be any party that is part of the set of channels to be updated. Following this, in phase *pre-setup*, $\mathcal{F}_{update}$ prepares all channels for the update by creating a synchronizing transaction $\mathsf{tx}^{\mathsf{ep}}$ per channel that can later be used to force all payments. In phase *setup*, $\mathcal{F}_{update}$ proceeds with preparing an intermediary state update for each channel. In this intermediary state, the payment can be enforced if any of the synchronizing transactions gets posted to $\mathcal{G}_{Ledger}$ and reverted after timeout T. Then, in phase *confirmation*, the updates to the intermediary states are performed via $\mathcal{F}_{Channel}$.

The functionality $\mathcal{F}_{update}$ proceeds to the *finalizing* phase iff all updates are successful and either the set of senders are honest or the simulator provided a valid signature from all dishonest senders for the synchronizing transactions. This is crucial because at this

point $\mathcal{F}_{update}$ can enforce the payment for honest receivers and only then it is safe to start finalizing. In the *finalizing* phase, all channels are finalized, i.e., updated to the state where the payment went through. If an update fails, $\mathcal{F}_{update}$ can utilize the synchronizing transaction to ensure that the payment is forced for honest receivers.

Further, the functionality checks each round if a synchronizing transaction $\mathsf{tx}^{\mathsf{ep}}$ was posted to $\mathcal{G}_{Ledger}$. This can be achieved by expecting the environment to pass the execution token to $\mathcal{F}_{update}$ each round. If it does not, $\mathcal{F}_{update}$ outputs an error the next time it gets the execution token. In case a synchronizing transaction is posted, $\mathcal{F}_{update}$ can force the payment on $\mathcal{G}_{Ledger}$. Similarly, a refund can be forced after T.

5.5.3 Informal security analysis

Here, we informally argue why the Thora protocol description shown in Section 5.4.2 achieves atomicity and strong value privacy as defined in Section 5.3.1.

Atomicity. We want to show that if there exist two channels with different update statuses, where each has at least one honest user, then the party deviating from the protocol loses the payment value in favor of the other (honest) channel end-point.

Assume that for two channels γ_i, γ_j, each with at least one honest user and with payment values α_i and α_j, γ_i is updated successfully, but γ_j is reverted. There are two possible cases as follows.

1. The final update in γ_i is done by $\gamma_i.\mathsf{sender}$ using $\mathsf{tx}_i^{\mathsf{trans}}$. If $\gamma_i.\mathsf{sender}$ has followed the protocol correctly, she should receive `confirmation-ok` message from all receivers, including $\gamma_j.\mathsf{receiver}$. So, $\gamma_j.\mathsf{receiver}$ has enough signatures to put $\mathsf{tx}_j^{\mathsf{ep}}$ on the ledger and force the payment. If $\gamma_i.\mathsf{sender}$ has finalized γ_i without receiving all `confirmation-ok` messages, she is deviating from the protocol at the cost of losing her funds to $\gamma_i.\mathsf{receiver}$. Also, if $\gamma_j.\mathsf{receiver}$ has sent `confirmation-ok` without having enough signatures or refuses to force the payment using $\mathsf{tx}_j^{\mathsf{ep}}$, she is deviating from the protocol at the cost of losing her funds to $\gamma_j.\mathsf{sender}$. None of the cases would affect others' security.
2. The payment in γ_i is forced via posting an enable payment transaction $\mathsf{tx}_k^{\mathsf{ep}}$ and $\mathsf{tx}_{i,k}^{\mathsf{p}}$ on the ledger. Thus, all other receivers, including $\gamma_j.\mathsf{receiver}$, can force the payment in their channels using $\mathsf{tx}_k^{\mathsf{ep}}$. Note that $\mathsf{tx}_k^{\mathsf{ep}}$ contains an output owned by $\gamma_j.\mathsf{receiver}$, otherwise, this user would not send `setup-ok` to other parties, including $\gamma_i.\mathsf{sender}$. If $\gamma_i.\mathsf{sender}$ continued the protocol without receiving all `setup-ok` messages, she is deviating from protocol at the cost of losing her funds. Also, if $\gamma_j.\mathsf{receiver}$ has sent `setup-ok` having incorrect $\mathsf{tx}_k^{\mathsf{ep}}$ or refuses to force the payment using $\mathsf{tx}_k^{\mathsf{ep}}$, she is deviating from the protocol at the cost of losing her funds to $\gamma_j.\mathsf{sender}$. None of the cases would affect others' security.

Strong value privacy. For an optimistic execution of the protocol, the value of payment α_i through each channel γ_i is only known to the sender and the receiver of this channel.

α_i is used only in $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. These transactions are exchanged between $\gamma_i.\mathsf{sender}$ and $\gamma_i.\mathsf{receiver}$ through secure and authenticated channels. If both parties are honest, the payment value is not visible to an adversary.

5.6 Evaluation

In this section, we analyze the performance of our construction. We conducted an asymptotic analysis to determine the number of transactions required on-chain and off-chain. We also built an implementation to evaluate the size of these transactions and to check the compatibility of the construction with Bitcoin's scripting functionalities. The implementation is open-source and the code is publicly available [Tho22]. Let n be the number of payment channels to be updated, which means that there are n possibly non-distinct senders and n possibly non-distinct receivers, and $m \in [0, n]$ be the number of channels in which parties do not agree to update off-chain, and therefore on-chain transactions are required to settle the dispute.

Number and size of transactions. In the honest case, Thora happens completely off-chain, requiring no on-chain cost. The (worst-case) on-chain overhead of the scheme is linear, requiring $2m + 1$ transactions to be posted on-chain. As shown in Table 5.2 and discussed below, this is in line with the state-of-the-art Bitcoin-compatible PCN protocols (e.g., Lightning Network and Blitz). In Thora, however, users are required to store a linear number of off-chain transactions per channel (which results in a quadratic number of total off-chain transactions), whereas the off-chain overhead for the existing Bitcoin-compatible PCN protocols is only constant per channel (or linear in total). We argue that this is a reasonable price to pay for supporting a larger class of off-chain applications, as (i) this increase does not lead to any extra on-chain fees and (ii) the size is small enough in practice to be easily handled even on mobile devices, as we show now.

The transaction $\mathsf{tx}^{\mathsf{ep}}$ is $141n + 160$ bytes large since it requires an output and a signature for each channel. Making use of Taproot's aggregated Schnorr signatures [Tap21], one can reduce the size of this transaction to $38n + 256$ bytes. This is achieved by eliminating n public keys (32 bytes) and signatures (70-72 bytes) from the redeem script in $\mathsf{tx}^{\mathsf{ep}}$, adding instead one Schnorr public key (32 bytes), which is the aggregation of public keys of one receiver and n senders, and one Schnorr signature (64 bytes).

Moreover, each channel requires n transactions tx^{p} (501 bytes each), one transaction tx^{r} (272 bytes), an input transaction to $\mathsf{tx}^{\mathsf{ep}}$ (224 bytes), a channel update of size 380 bytes for initiating the update, and another one of size 337 bytes for finalizing the update. For the whole protocol execution, this leads to an off-chain storage overhead of $539n + 1469$ bytes per channel as we plot in Figure 5.6. For example, even when updating $n = 100$ channels, the off-chain transaction overhead is only around 55KB per channel, or around 5.5MB are exchanged in total.

Collateral. Because the success of the update depends on the global event $\mathsf{tx}^{\mathsf{ep}}$, Thora manages a constant collateral lock time. For the payment protocols LN [PD16] and

AMHL [MMS+19], this collateral is instead linear in the number of channels, as they require a growing timelock for each channel to propagate the preimage required for unlocking. In PT [JLT21], the time is logarithmic due to the underlying tree-based structure. Finally, Blitz [AMSKM21], Sprites [MBB+19b], and AMCU [EMSM19] achieve also constant collateral, at the price of various security, expressiveness, and compatibility trade-offs (cf. Tables 5.1 and 5.2).

Computational overhead. Computationally, the protocol needs to create and verify transactions (mostly string operations) and handle signatures. In particular, the computational overhead is dominated by computing and verifying signatures. Each sender needs to sign up to $2n + 2$ transactions, more specifically the channel update transaction $\mathsf{tx}^{\mathsf{state}}$, one force refund transaction tx^{r} which they need only in case of dispute, n force payment transactions tx^{p} for their receiver neighbors, and n transactions $\mathsf{tx}^{\mathsf{ep}}$, one for each receiver. Each receiver signs up to $n + 2$ transactions, i.e., the channel update transaction $\mathsf{tx}^{\mathsf{state}}$, one force payment transaction tx^{p} which they need only in case of dispute, and their own transaction $\mathsf{tx}^{\mathsf{ep}}$. In our implementation, the time required for creating and verifying one signature is about 30ms on average.

On-chain comparison with LN and Blitz. In Table 5.3, we compare the on-chain costs of Thora with LN and Blitz, the two state-of-the-art solutions for path-based payments. We assume that Thora is used to conduct such a payment and focus on the on-chain load on the blockchain together with the associated fees, which we calculate using the current price of Bitcoin in USD [Bit22b] and the current average fee per bytes [Bit22d] (February 2022). When all parties are honest, both protocols are executed completely off-chain, and no transaction is required to appear on the ledger, thus here we are interested in the case where parties need to force either the payment or the refund.

Thora and Blitz have similar message costs, just the cost for the payment and refund transactions are inverted, which corresponds to the fact that one adopts the pay-unless-revoke paradigm and the other one the revoke-unless-pay paradigm. The size of the channel state transaction holding the update contract (370 bytes) is the same in all three constructions, due to our usage of P2SH addresses. The size of the payment transaction in LN is 451 bytes, and the size of the refund is 302 bytes. The main difference between the on-chain overhead of these two protocols is $\mathsf{tx}^{\mathsf{ep}}$ in Thora. In the case of forced payments, in addition to one tx^{p} per channel, one $\mathsf{tx}^{\mathsf{ep}}$ in total has to be posted to the ledger to enable payments in all channels. This overhead is present in the Blitz refund case. Aside from this, the on-chain fees of Thora are similar to those for LN (the payment transaction is 6% more expensive, while the refund transaction is 6% cheaper). A difference to LN and similarity to Blitz is, that the user posting $\mathsf{tx}^{\mathsf{ep}}$ in Thora (or the equivalent transaction in Blitz) loses $(n-1) \cdot \epsilon$ coins. In Bitcoin, outputs cannot hold 0 coins, therefore ϵ is chosen to be the smallest possible value, e.g., for P2WPKH outputs this is currently 294 satoshis (roughly 0.06 USD). This cost is not present in LN.

Table 5.2: Asymptotic comparison of current solutions, with n being the number of channels.

	Collateral	# tx (on-chain)	# tx (off-chain)
LN [PD16]	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AMHL [MMS+19]	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
AMCU [EMSM19]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
PT [JLT21]	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Blitz [AMSKM21]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sprites [MBB+19b]	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Thora	$\Theta(1)$	$\Theta(n)$	$\Theta(n^2)$

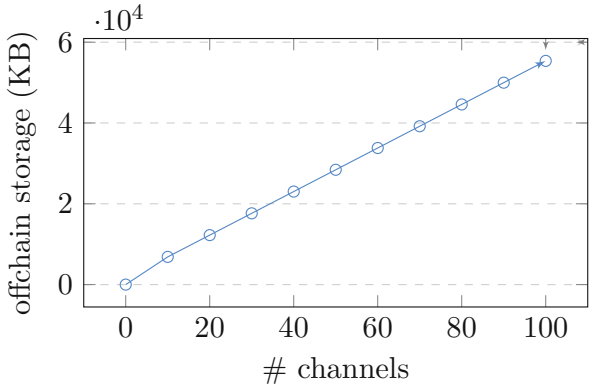

Figure 5.6: Per-channel off-chain storage overhead for varying number of synchronized channels.

Table 5.3: On-chain overhead and cost comparison of LN, Blitz and Thora. n is the number of channels and $m \in [0, n]$ is the number of disputed channels.

Overhead	LN (Bytes || USD)	Blitz (Bytes || USD)	Thora (Bytes || USD)
Payment transaction	$821m$ || $1.50m$	$642m$ || $1.17m$	$871m$ || $1.59m$
Refund transaction	$672m$ || $1.23m$	$871m$ || $1.59m$	$642m$ || $1.17m$
Cost of enforcing pay/refund	0	$257 + 35n$ || $0.47 + 0.06n$	$256 + 36n$ || $0.47 + 0.06n$

5.7 Applications

Most of the existing PCN solutions only support payments from one sender to one receiver and these are to be connected by a path of open channels. This limitation prevents the design of applications with multiple senders or multiple receivers, or those involving payments through two or more distinct PCNs sharing the same blockchain. We show below how Thora overcomes these limitations.

Mass payments. Mass payments can be used by entities that need to perform a high volume of payments. Suppose that a single entity S wants to pay multiple recipients

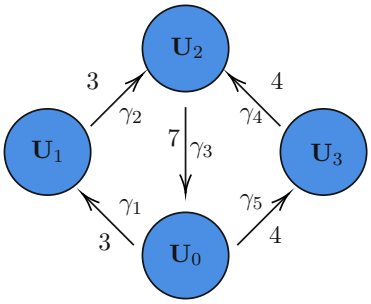

Figure 5.7: An example of rebalancing with 4 users and 5 channels. Each user holds the same coins after the rebalancing as before, but distribution of coins through channels is changed in order to refund depleted channels. In this case, rebalancing cannot be conducted using a single path-formed payment without using a channel more than once.

$R_1, R_2, ..., R_n$ simultaneously, with corresponding values $\alpha_1, \alpha_2, ..., \alpha_n$. Here, atomicity can be highly desirable as it guarantees that either all payments are performed correctly or the sender is refunded. For simplicity, we assume that S has a direct channel γ_i to each receiver R_i. The sender S can use Thora with the input of the update set $U := \{(\gamma_1, \alpha_1), (\gamma_2, \alpha_2), ..., (\gamma_n, \alpha_n)\}$ to perform a mass payment in an atomic and off-chain way. Going one step further, the sender does not need to be directly connected to all receivers but instead can set up updates via some intermediaries. A special case of this is when one sender wants to atomically pay one receiver over multiple paths at once, e.g., when the balance of one path is not sufficient. This is known as *atomic multi-path payment* [Ato22] and can be achieved with Thora.

Rebalancing. In a bidirectional channel, when payments in one direction are more frequent than in the other direction, the channel becomes skewed and is eventually reduced to a unidirectional channel. Users can close the channel and create a new channel with fresh balances, but for that, they need to post some transactions to the blockchain. Alternatively, if there exists a path of channels between the two users that wish to rebalance their channel, they can leverage a payment through this path to replenish the depleted channel. This can be more efficient if there are multiple users on the path who wish to rebalance their channels. However, as the length of the path grows, refunding becomes more expensive in terms of fees and collateral [EMSM19, KG17].

Moreover, in some cases, rebalancing is performed through more complex topologies, where (i) a single path payment does not suffice without using certain channels more than once, see Figure 5.7, or (ii) rebalancing can be made more efficient by making use of the *canceling out effect*, as shown in [APS+22]. In the example of Figure 5.7, users hold the same amount of coins after the payments as before, but the distribution of coins in the channels is changed. We can perform rebalancing in this case by initiating Thora with the input of the update set $\{(\gamma_1, 3), (\gamma_2, 3), (\gamma_3, 7), (\gamma_4, 4), (\gamma_5, 4)\}$. The set of senders and receivers is defined based on the direction of the payment in each channel.

Transaction aggregation. Suppose S_0 wants to pay 5 coins to R_1 and S_1 wants to pay 5 coins to S_0, however, there are only channels between S_0 and R_0 and between S_1 and R_1. A more generalized version of this problem was introduced as *transaction aggregation* in [TYA+22] along with a construction that uses Thora as a building block, which solves this problem.

Crowdfunding. This application is similar to mass payments, but reversed. We have multiple senders $S_0, S_1, ..., S_n$ who want to fund one single receiver R in an atomic way. In such a case, each sender S_i may want to pay α_i coins to the receiver only when there is a guarantee that all other senders will pay their funds in the same way. Analogous to previous cases, we can use Thora to perform trustless and off-chain crowdfunding by including all involved channels and corresponding payment values in the update set.

5.8 Discussion

Enhancing privacy. In the case of a dispute when one $\mathsf{tx}^{\mathsf{ep}}$ appears on the ledger, users can decide to perform honest updates (Section 5.3) and to post no transaction to the ledger. In this way, they can still preserve the privacy of payment values and save the cost of transaction fees. However, because $\mathsf{tx}^{\mathsf{ep}}$ includes outputs to all receivers, receivers' identities are revealed publicly when $\mathsf{tx}^{\mathsf{ep}}$ is posted.

To enhance privacy, we can use stealth addresses [VS18]. On a high level, instead of existing addresses, receivers can generate fresh addresses for other receivers, and create $\mathsf{tx}^{\mathsf{ep}}$ using new addresses. Thus, if any $\mathsf{tx}^{\mathsf{ep}}$ is posted to the ledger and the two channel users decide to update the channel honestly, their identities will stay private from all parties not involved in the protocol. For more details on stealth addresses, we refer the reader to Appendix D.1.

Accountability. Thora guarantees strong value privacy for off-chain payments. However, in some applications, users may have an interest in accounting payments instead of privacy. For instance, in the crowdfunding application, suppose that all senders have planned to fund the receiver entity with an identical value. Here, the users want to be sure all updates are consistent with the agreed payment value. In this case, the senders can use signed versions of $\mathsf{tx}^{\mathsf{state}}$ and the set of tx^{p} as receipts and prove their correct behavior.

Communication and computation complexity. As previously discussed, parties have to exchange off-chain messages with each other (i.e., $\mathsf{tx}^{\mathsf{ep}}$ and signatures), which leads to quadratic communication overhead. By extending the role of dealer to a user whom all parties send these messages and who aggregates the signatures, one could asymptotically reduce the number of signatures that each party has to handle from linear to constant since only the aggregated signature is sent instead of every individual one. Note that, despite the resulting gain, the size of the transactions is, technically speaking, still quadratic from an asymptotical point of view, because $\mathsf{tx}^{\mathsf{ep}}$ has a linear number of outputs and there is one for every channel.

Race condition. When a receiver posts $\mathsf{tx}^{\mathsf{ep}}$, it will appear in the ledger after at most Δ rounds. According to Section 5.3, we put a timelock of $t_c + \Delta$ on outputs of a $\mathsf{tx}^{\mathsf{ep}}$ to give enough time to users to close their channels and post tx^{p}. Thus, for a rational receiver, the latest possible time to publish $\mathsf{tx}^{\mathsf{ep}}$ is $T - 3\Delta - t_c$, so that it is accepted at $T - 2\Delta - t_c$ and the timelock of the outputs runs out at $T - \Delta$. This ensures that the payment tx^{p} has precedence over the refund tx^{r}. However, if a receiver posts $\mathsf{tx}^{\mathsf{ep}}$ after $T - 3\Delta - t_c$ and before $T - 2\Delta - t_c$, the timelock on the outputs of $\mathsf{tx}^{\mathsf{ep}}$ could run out just before T, at which point the refunds tx^{r} become possible. Now, there is a potential race between the payments and the refunds. In particular, there is a chance that one receiver can post tx^{p} just before T, and in another channel, a sender might post a refund.

Of course, this behavior is irrational since the receiver puts her balance and possibly the one of other malicious receivers at risk, as other channels with honest receivers will have already either updated honestly or posted their $\mathsf{tx}^{\mathsf{ep}}$ before $T - 3\Delta - t_c$. If interested, we can anyway prevent this race condition caused by irrational receivers by changing the spending condition of $\mathsf{tx}^{\mathsf{in}}$. In more detail, each receiver R sets the condition of her $\mathsf{tx}^{\mathsf{in}}$ as follows: $(\mathsf{MultiSig}(R, S_1, S_2, ..., S_n) \wedge \mathsf{RelTime}(\Delta)) \vee (\mathsf{AbsTime}(T - 3\Delta - t_c))$, where S_i is the sender of channel γ_i. According to the new condition, the receiver is forced to post $\mathsf{tx}^{\mathsf{ep}}$ before $T - 5\Delta - t_c$, because otherwise, any party, e.g., also miners, can spend $\mathsf{tx}^{\mathsf{in}}$ and prevent forced payments. This mechanism is similar to the one adopted in Blitz [AMSKM21].

5.9 Conclusion

In this work, we presented Thora, the first Bitcoin-compatible multi-channel update protocol that guarantees atomicity of payments without restrictions on the channel topology. Moreover, Thora enables channel owners to keep their payment value private.

We defined an ideal functionality to model the security and privacy notations of interest and showed that Thora is a secure realization thereof within the *Global Universal Composability* framework. Further, we evaluated the performance and showed that the collateral is constant and independent of the number of channels. Our construction does not require Turing-complete smart contracts and can be implemented on top of any blockchain that supports time-locks and signatures in its scripting language.

An interesting direction of future work is exploring the possibility to extend Thora to achieve a threshold atomicity property in generic channel networks. For instance, a k-threshold atomicity holds, if at least k channels are updated successfully or else, all channels are reverted to the initial state. This extension can further widen the range of practical applications of Thora payments. Other venues of future research are interoperability, exploring how to refine Thora in order to support atomic channel updates over different blockchains, and optimizing Thora in terms of storage and communication for more specific network topologies.

Acknowledgements

The work was partially supported by CoBloX Labs, by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement 771527-BROWSEC), by the Austrian Science Fund (FWF) through the projects PROFET (grant agreement P31621) and the project W1255-N23, by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694), the COMET K1 SBA and COMET K1 ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT).

CHAPTER 6

Bitcoin-Compatible Virtual Channels

Abstract

Current permissionless cryptocurrencies such as Bitcoin suffer from a limited transaction rate and slow confirmation time, which hinders further adoption. Payment channels are one of the most promising solutions to address these problems, as they allow the parties of the channel to perform arbitrarily many payments in a peer-to-peer fashion while uploading only two transactions on the blockchain. This concept has been generalized into payment channel networks where a path of payment channels is used to settle the payment between two users that might not share a direct channel between them. However, this approach requires the active involvement of each user in the path, making the system less reliable (they might be offline), more expensive (they charge fees per payment), and slower (intermediaries need to be actively involved in the payment). To mitigate this issue, recent work has introduced the concept of virtual channels (IEEE S&P'19), which involve intermediaries only in the initial creation of a bridge between payer and payee, who can later on independently perform arbitrarily many off-chain transactions. Unfortunately, existing constructions are only available for Ethereum, as they rely on its account model and Turing-complete scripting language. The realization of virtual channels in other blockchain technologies with limited scripting capabilities, like Bitcoin, was so far considered an open challenge.

In this work, we present the first virtual channel protocols that are built on the UTXO model and require a scripting language supporting only a digital signature scheme and a timelock functionality, being thus backward compatible with virtually every cryptocurrency, including Bitcoin. We formalize the security properties of virtual channels as an ideal functionality in the Universal Composability framework and prove that our protocol constitutes a secure realization thereof. We have prototyped and evaluated our

protocol on the Bitcoin blockchain, demonstrating its efficiency: for n sequential payments, they require an off-chain exchange of 9+2n transactions or a total of 3524+695n bytes, with no on-chain footprint in the optimistic case. This is a substantial improvement compared to routing payments in a payment channel network, which requires 8n transactions with a total of 3026n bytes to be exchanged.

This chapter presents the results of a collaboration with Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi, which was published at the 42nd IEEE Symposium on Security and Privacy in 2021 under the title "Bitcoin-Compatible Virtual Channels". I am responsible for writing a proof-of-concept implementation, deploying and testing the scheme on the Bitcoin testnet, and designing and conducting the experiments and measurements. Further, I contributed to the write-up of the paper. The paper presents two virtual channel constructions (with and without validity). Oğuzhan Ersoy and I are responsible for designing virtual channels with validity. Siavash Riahi and Andreas Erwig are mainly responsible for designing virtual channels without validity and writing the security proofs of that protocol. Kristina Hostáková and Siavash Riahi are responsible for modeling the security properties and writing the ideal functionality. Pedro Moreno-Sanchez, Sabastian Faust and Matteo Maffei were the general advisors and contributed with continuous feedback.

6.1 Introduction

Permissionless cryptocurrencies such as Bitcoin [Nak09] have spurred increasing interest over the last years, putting forward a revolutionary, from both a technical and economical point of view, payment paradigm. Instead of relying on a central authority for transaction validation and accounting, Bitcoin relies on its core on a decentralized consensus protocol for these tasks. The consensus protocol establishes and maintains a distributed ledger that tracks every transaction, thereby enabling public verifiability. This approach, however, severely limits the transaction throughput and confirmation time, which in the case of Bitcoin is around ten transactions per second, and confirmation of an individual transaction can take up to 60 minutes. This is in stark contrast to central payment providers that offer instantaneous transaction confirmation and support orders of magnitude higher transaction throughput. These scalability issues hinder permissionless cryptocurrencies such as Bitcoin from serving a growing base of payments.

Within other research efforts [GMSR+20, ZABZ+21, BSA+17], payment channels [Bit18] have emerged as one of the most promising scalability solutions. The most prominent example that is currently deployed over Bitcoin is the so-called Lightning network [PD16], which at the time of writing hosts deposits worth more than 60M USD. A payment channel enables an arbitrary number of payments between users while committing only two transactions onto the blockchain. In a bit more detail, a payment channel between Alice and Bob is first created by a single on-chain transaction that deposits Bitcoins into a multi-signature address controlled by both users. The parties additionally ensure that they can get their Bitcoins back at a mutually agreed expiration time. They can then pay to each other (possibly many times) by exchanging authenticated off-chain messages

that represent an update of their share of coins in the multi-signature address. The payment channel is finally closed when a user submits the last authenticated distribution of Bitcoins to the blockchain (or after the channel has expired).

Interestingly, it is possible to leverage a path of opened payment channels from the sender to the receiver with enough capacity to settle their payments off-chain, thereby creating a payment channel network (PCN) [PD16, MMSK+17]. Assume that Alice wants to pay Bob, and they do not have a payment channel between each other but rather are connected through an intermediary user Ingrid. Upon a successful off-chain update of the payment channel between Alice and Ingrid, the latter would update her payment channel with Bob to make the overall transaction effective. The key challenge is how to perform the sequence of updates atomically in order to prevent Ingrid from stealing the money from Alice without paying Bob. The standard technique for constructing PCNs requires the intermediary (e.g., Ingrid in the example from above) to be actively involved in each payment. This has multiple disadvantages, including (i) making the system less reliable (e.g., Ingrid might have to go offline), (ii) increasing the latency of each payment, (iii) augmenting its costs since each intermediary charges a fee per transaction, and (iv) revealing possibly sensitive payment information to the intermediaries [NFSD20, TMM20, KYP+21].

An alternative approach for connecting multiple payment channels was introduced by Dziembowski et al. [DEFM19]. They propose the concept of *virtual channels* – an off-chain protocol that enables direct off-chain transactions without the involvement of the intermediary. Following our running example, a virtual channel can be created between Alice and Bob using their individual payment channels with Ingrid. Ingrid must collaborate with Alice and Bob only to create such a virtual channel, which can then be used by Alice and Bob to perform arbitrarily many off-chain payments without involving Ingrid. Virtual channels offer strong security guarantees: each user does not lose money even if the others collude. A salient application of virtual payment channels is so-called payment hubs [DEFM19]. Since establishing a payment channel requires a deposit and active monitoring, the number of channels a user can establish is limited. With payment hubs [DEFM19], users have to establish just one payment channel with the hub and can then dynamically open and close virtual channels between each other on demand. Interestingly, since in a virtual channel, the hub is not involved in individual payments, even transactions worth fractions of cents can be carried out with low latency.

The design of secure virtual channels is very challenging since, as previously mentioned, it has to account for all possible compromise and collusion scenarios. For this purpose, existing virtual channel constructions [DEFM19] require smart contracts programmed over an expressive scripting language and the account model, as supported in Ethereum. This significantly simplifies the construction since the deposit of a channel, and its distribution between the end-points are stored in memory and can programmatically be updated. On the downside, however, these requirements currently limit the deployment of virtual channels to Ethereum.

It was an *open question* until now if virtual channels could be implemented at all in

UTXO-based cryptocurrencies featuring only a limited scripting language, like Bitcoin and virtually all other permissionless cryptocurrencies. We believe that answering this question is important for several reasons. First, by limiting the trusted computing base (i.e., the scripting functionality supported by the underlying blockchain), we reduce the on-chain complexity of the virtual channel protocol. As bugs in smart contracts are manifold and notoriously hard to fix, our construction eliminates an additional attack vector by moving the complexity to the protocol level (rather than on-chain as in the construction from [DEFM19]). Second, investigating the minimal functionality that is required by the underlying ledger to support complex protocols is scientifically interesting. One may view this as a more general research direction of building a lambda calculus for off-chain protocols. Concretely, our construction shows that virtual channels can be built with stateless scripts, while earlier constructions required stateful on-chain computation. Finally, from a practical perspective, our construction can be integrated into the Lightning Network (the by far most prominent PCN), and thus our solution can offer the benefits of virtual payment channels/hubs to a broad user base.

6.1.1 Our contributions

In this work, we develop the *first* protocols for building virtual channel hubs over cryptocurrencies that support limited scripting functionality. Our construction requires only digital signatures and timelocks, which are ubiquitously available in cryptocurrencies and well characterized. We also provide a comprehensive formal analysis of our constructions and benchmarks of a prototype implementation. Concretely, our contributions are summarized below.

- We present the first protocols for virtual channel hubs that are built for the UTXO model and require a scripting language supporting only digital signature verification and timelock functionality, being thus compatible with virtually every cryptocurrency, including Bitcoin. Since in the Lightning network currently only 10 supernodes are involved in more than 25% of all channels, our technique can be used to reduce the load on these nodes, and thereby help to reduce latency.

- We offer two constructions that differ on whether (i) the virtual channel is guaranteed to stay off-chain for an encoded validity period, or (ii) the intermediary Ingrid can decide to offload the virtual channel (i.e., convert it into a direct channel between Alice and Bob), thereby removing its involvement in it. These two variants support different business and functionality models, analogous to non-preemptible and preemptible virtual machines in the cloud setting, with Ingrid playing the role of the service provider.

- We formalize the security properties of virtual channels as an ideal functionality in the UC framework [Can01], and prove that our protocols constitute a secure realization thereof. Since our virtual channels are built in the UTXO model, our ideal functionality and formalization significantly differs from earlier work [DEFM19].

- We evaluate our protocol over two different PCN constructions, the Lightning Network (LN) [PD16] and Generalized channels (GC) [AEE+21], which extend LN channels to

support functionality other than one-to-one payments. We show that for virtual channels on top of GC, n sequential payment operations require an off-chain exchange of $9 + 2 \cdot n$ transactions or a total of $3524 + 695 \cdot n$ bytes, as compared to $8 \cdot n$ transactions or $3026 \cdot n$ bytes when Ingrid routes the payment actively through the PCN. This means a virtual channel is already cheaper if two or more sequential payments are performed. For virtual channels over LN, n transactions require an off-chain exchange of $6292 + 2824 \cdot n$ bytes, compared to $4776 \cdot n$ bytes when routed through an intermediary. We have interacted with the Bitcoin blockchain to store the required transactions, demonstrating the compatibility of our protocol.

To summarize, for the first time in Bitcoin, we enable off-chain payments between users connected by payment channels via a hub without requiring the continuous presence of any intermediary. Hence, our solution increases the reliability and, at the same time, reduces the latency and costs of Bitcoin PCNs.

6.2 Background

In this section, we first introduce notation and preliminaries on UTXO-based blockchains. We then overview the basics of payment and virtual channels, referring the reader to [Ant14, MMSK+17, MMS+19, DEFM19] for further details. We finally discuss the main technical challenges one needs to overcome when constructing Bitcoin-compatible virtual channels.

6.2.1 UTXO-based blockchains

We adopt the notation for UTXO-based blockchains from [AEE+21], which we shortly review below.

Attribute tuples. Let T be a tuple of values, which we call in the following *attributes*. Each attribute in T is identified by a unique keyword, e.g., attr and referred to as $T.\mathsf{attr}$.

Outputs and transactions. We focus on blockchains based on the Unspent Transaction Output (UTXO) model, such as Bitcoin. In the UTXO model, coins are held in *outputs* of transactions. Formally, an output θ is an attribute tuple $(\theta.\mathsf{cash}, \theta.\varphi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins associated with the output and $\theta.\varphi$ denotes the conditions that need to be satisfied in order to spend the output. The condition $\theta.\varphi$ can contain any set of operations (also called scripts) supported by the considered blockchain. We say that a user P controls or owns an output θ if $\theta.\varphi$ contains only a signature verification w.r.t. the public key of P.

In a nutshell, a *transaction* in the UTXO model, maps one or more existing outputs to a list of new outputs. The existing outputs are called *transaction inputs*. Formally, a transaction tx is an attribute tuple and consists of the following attributes $(\mathsf{tx.txid}, \mathsf{tx.input}, \mathsf{tx.output}, \mathsf{tx.TimeLock}, \mathsf{tx.Witness})$. The attribute $\mathsf{tx.txid} \in \{0,1\}^*$ is called the identifier of the transaction. The identifier is calculated as $\mathsf{tx.txid} := \mathcal{H}([\mathsf{tx}])$, where $\mathcal{H}$ is a hash

function which is modeled as a random oracle and $[\mathsf{tx}]$ is the *body of the transaction* defined as $[\mathsf{tx}] := (\mathsf{tx.input}, \mathsf{tx.output}, \mathsf{tx.TimeLock})$. The attribute $\mathsf{tx.input}$ is a vector of strings which identify the inputs of tx. Similarly, the outputs of the transaction $\mathsf{tx.output}$ is the vector of new outputs of the transaction tx. The attribute $\mathsf{tx.TimeLock} \in \mathbb{N} \cup \{0\}$ denotes the absolute time-lock of the transaction, which intuitively means that transaction tx will not be accepted by the blockchain before the round defined by $\mathsf{tx.TimeLock}$. The time-lock is by default set to 0, meaning that no time-lock is in place. Lastly, $\mathsf{tx.Witness} \in \{0,1\}^*$ called the transaction's witness, contains the witness of the transaction that is required to spend the transaction inputs.

We use charts in order to visualize the transaction flow in the rest of this work. We first explain the notation used in the charts and how they should be read. Transactions are shown using rectangles with rounded corners. Double-edge rectangles are used to represent transactions that are already published on the blockchain. Single-edge rectangles are transactions that could be published on the blockchain, but they are not yet. Each transaction contains one or more boxes (i.e., with squared corners) that represent the outputs of that transaction. The amount of coins allocated to each output is written inside the output box. In addition, the output condition is written on the arrow coming from the output.

In order to be concise, we use the following abbreviations for the frequently used conditions. Most outputs can only be spent by a transaction that is signed by a set of parties. In order to depict this condition, we write the public keys of all these parties *below* the arrow. We use the command `One–Sig` and `Multi–Sig` in the pseudocode. Other additional spending conditions are written *above* the arrow. The output script can have a relative time lock, i.e., a condition that is satisfied if and only if at least t rounds are passed since the transaction was published on the blockchain. We denote this output condition by writing the string "$+t$" *above* the arrow (and `CheckRelative` in the pseudocode). In addition to relative time locks, an output can also have an absolute time lock, i.e., a condition that is satisfied only if t rounds elapsed since the blockchain was created and the first transaction was posted on it. We write the string "$> t$" *above* the arrow for this condition. Lastly, an output's spending condition might be a disjunction of multiple conditions. In other words it can be written as $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ for some $n \in \mathbb{N}$ where φ is the output script. In this case, we add a diamond shape to the corresponding transaction output. Each of the subconditions φ_i is then written above a separate arrow. An example is given in Figure 6.1.

6.2.2 Payment channels

A payment channel enables arbitrarily many transactions between users while requiring only two on-chain transactions. The first step when creating a payment channel is to deposit coins into an output controlled by two users. Once the money is deposited, the users can authorize new balance updates in a peer-to-peer fashion while having the guarantee that all coins are refunded at a mutually agreed time. In a bit more detail, a

Figure 6.1: (Left) Transaction tx is published on the blockchain. The output of value x_1 can be spent by a transaction signed w.r.t. pk_B after round t_2, and the output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_3 rounds passed since tx was accepted by the blockchain. (Right) Transaction tx' is not published on the ledger. Its only output, which is of value x, can be spent by a transaction whose witness satisfies the output condition $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

payment channel has three operations: *open*, *update* and *close*. We necessarily keep the description short and refer to [GMSR+20, AEE+21] for further reading.

Open. Assume that Alice and Bob want to create a payment channel with an initial deposit of x_A and x_B coins, respectively. For that, Alice and Bob agree on a *funding transaction* (that we denote by tx^{f}) that sets as inputs two outputs controlled by Alice and Bob holding x_A and x_B coins respectively, and transfers them to an output controlled by both Alice and Bob. When tx^{f} is added to the blockchain, the payment channel is effectively open.

Update. Assume now that Alice wants to pay $\alpha \leq x_A$ coins to Bob. For that, they create a new *commit transaction* $\mathtt{TX}_{\mathsf{c}}$ representing the commitment from both users to the new balance of the channel. The commit transaction spends the output of tx^{f} into two new outputs: (i) one holding $x_A - \alpha$ coins controlled by Alice; and (ii) the other holding $x_B + \alpha$ coins controlled by Bob. Finally, parties exchange signatures on the commit transaction, which serve as valid witnesses for tx^{f}. At this point, Alice (resp. Bob) could add $\mathtt{TX}_{\mathsf{c}}$ to the blockchain. Instead, they keep it locally in their memory and overwrite it when they agree on another commitment transaction $\overline{\mathtt{TX}}_{\mathsf{c}}$ representing a newer balance of the channel. This, however, leads to the problem that there exist several commitment transactions that can possibly be added to the blockchain. Since all of them are spending the same output, only one can be accepted by the blockchain. Since it is impossible to prevent a malicious user from publishing an outdated commit transaction, payment channels require a mechanism that punishes such malicious behavior. This mechanism is typically called *revocation* and enables that an honest user can take all the coins locked in the channel if the dishonest user publishes an outdated commitment transaction.

Close. Assume finally that Alice and Bob no longer wish to use the channel. Then, they can collaboratively close the channel by submitting the last commitment transaction $\overline{\mathtt{TX}}_{\mathsf{c}}$ that they have agreed on to the blockchain. After it is accepted, the coins initially locked at the channel creation via tx^{f} are redistributed to both users according to the last agreed balance. As aforementioned, if one of the users submits an outdated commitment transaction instead, the counterparty can punish the former through the revocation mechanism.

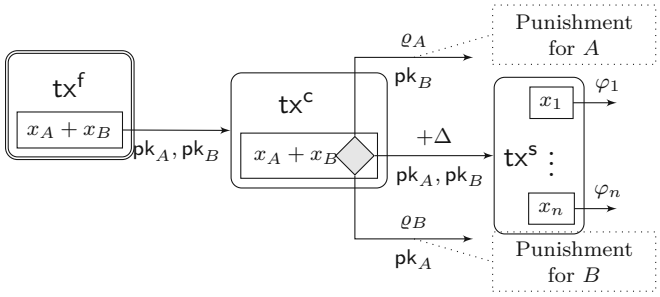

Figure 6.2: A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. The value of Δ upper bounds the time needed to publish a transaction on a blockchain. The condition ϱ_A represents the verification of A' revocation secret and ϱ_B represents the verification of B' revocation secret.

The Lightning Network [PD16] defines the state-of-the-art payment channel construction for Bitcoin.

6.2.3 Generalized channels

The recent work of Aumayr et al. [AEE+21] proposes the concept of *generalized channels*. Generalized channels improve and extend payment channels (see Figure 6.2 for details) in two ways. First, they extend the functionality of payment channels by offering off-chain execution of any script that is supported by the underlying ledger. Hence, one may view generalized channels as state channels for blockchains with restricted scripting functionality. Second, and more important for our work, generalized channels significantly improve the on-chain and off-chain communication complexity. More concretely, this efficiency improvement is achieved by introducing a so-called *split transaction* (that we denote as $\mathtt{TX}_\mathsf{s}$) along with a *punish-then-split* paradigm. In contrast to regular payment channels that require one revocation process per output in the commit transaction, the punish-then-split approach decouples the revocation process from the number of outputs in the commit transaction. This allows moving from revocation for each output to a single revocation for the entire channel. As shown in Figure 6.2, the commit transaction ($\mathtt{TX}_\mathsf{c}$) is only responsible for the punishment, while the split transaction ($\mathtt{TX}_\mathsf{s}$) holds the actual outputs of the channel.

The efficiency of generalized channels is further improved since they only require a single commit transaction per channel. This is in contrast to the payment channels used by Lightning, which require two distinct commit transactions for each channel user. We will discuss in Section 6.3.4 why the punish-then-split paradigm (and requiring only one commit transaction) is useful in order to improve the efficiency of our virtual channels for Bitcoin.

To simplify terminology, we will use the term *ledger channel* for all channels that are funded directly over the blockchain.

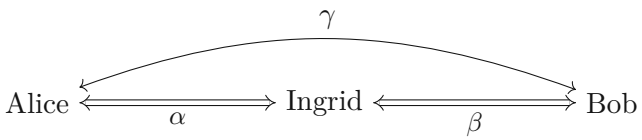

Figure 6.3: A virtual channel γ built over ledger channels α, β.

6.2.4 Channel Networks

The aforementioned payment and generalized channels allow two parties to issue transactions between each other while having to communicate with the blockchain only during the creation and closure of the channel. This on-chain communication can further be reduced by using *channel networks*. **Payment Channel Networks (PCNs).** A PCN is a protocol that allows parties to connect multiple ledger channels to form a payment channel network. In this network, a sender can route a payment to a receiver as long as both parties are connected by a path in the network. Suppose that Alice and Bob are not directly connected via a ledger channel, but instead, both maintain a channel with an intermediary party (Ingrid). In a nutshell, Alice can pay Bob by sending her coins to Ingrid who then forwards them in her ledger channel to Bob. Importantly, the protocol must achieve atomicity, i.e., either both transfers from Alice to Ingrid and from Ingrid to Bob happen, or neither of them goes through. Current PCNs such as the Lightning network use the HTLC technique (hash-time-lock transaction), which comes with several drawbacks as mentioned in the introduction: (i) *low reliability* because the success of payments relies on Ingrid being online; (ii) *high latency* as each payment must be routed through Ingrid; (iii) *high-cost* as Ingrid may charge a fee for each payment between Alice and Bob; and (iv) *low privacy* as Ingrid can observe each payment that happens between Alice and Bob. To mitigate these issues, virtual channels have been proposed.

Virtual Channels. An alternative solution to connect two payment channels with each other is offered by the concept of *virtual channels* [DEFM19]. Virtual channels allow Alice and Bob to send payments between each other without the involvement of the intermediary Ingrid. In some sense, they thus mimic the functionality offered by ledger channels, with the difference that they are not created directly over the blockchain but instead over two ledger channels. More concretely, as shown in Figure 6.3, a virtual channel γ between Alice and Bob with intermediary Ingrid is constructed on top of two ledger channels α and β. Ingrid is required to participate in the initial creation and final closing of the virtual channel. But importantly, Ingrid is not involved in any balance updates that occur in the virtual channel. This overcomes the four drawbacks mentioned above. While these advantages over PCNs make virtual channels an attractive off-chain solution, their design is far from trivial. Previous work showed how to construct virtual channels over a ledger that supports Turing complete smart contracts [DEFM19, DFH18, DEF+19b]. The smart contract acts in the protocol as a trust anchor that parties can fall back to in case of malicious behavior. Through a rather complex protocol and careful smart

contract design, existing virtual channel constructions guarantee that honest parties in the virtual channel will always get the coins they rightfully own. Unfortunately, most cryptocurrencies (including Bitcoin) do not offer Turing complete smart contracts, and hence the constructions from prior work cannot be used. In this work, we present a novel construction of virtual channels that makes only minimal assumptions on the underlying scripting functionality offered by the ledger.

6.3 Virtual Channels

In this section, we first give some notation before presenting the necessary properties for virtual channels and discussing design challenges. Finally, we present our protocol.

6.3.1 Definitions

We briefly recall some notation and definition for generalized channels [AEE+21] and extend the definition to generalized virtual channels. In order to make the distinction between the two types of channels clearer, we call the former generalized *ledger* channel (or ledger channels for short).

A *generalized ledger channel* as defined in [AEE+21] is a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob} \in \mathcal{P}$ are the identities of the parties using the channel, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is a finite precision real number that represents the total amount of coins locked in this channel and $\gamma.\mathsf{st} = (\theta_1, \ldots, \theta_n)$ is the state of the channel. This state is composed of a list of *outputs*. Recall that each output θ_i has two attributes: the output value $\theta_i.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ and the output condition $\theta_i.\varphi\colon \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \to \{0,1\}$. For convenience, we define a set $\gamma.\mathsf{users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and a function $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{users} \to \gamma.\mathsf{users}$, which on input $\gamma.\mathsf{Alice}$ outputs $\gamma.\mathsf{Bob}$ and on input $\gamma.\mathsf{Bob}$ returns $\gamma.\mathsf{Alice}$.

A generalized *virtual channel* (or for short virtual channel) is defined as a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{subchan}, \gamma.\mathsf{fee}, \gamma.\mathsf{val})$. The attributes $\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st}$ are defined as in the case of ledger channels. The additional attribute $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ denotes the identity of the *intermediary* of the virtual channel γ. The set $\gamma.\mathsf{users}$ and the function $\gamma.\mathsf{otherParty}$ are defined as before. Additionally, we also define the set $\gamma.\mathsf{users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}\}$. The attribute $\gamma.\mathsf{subchan}$ is a function mapping $\gamma.\mathsf{users}$ to a channel identifier; namely, the value $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ refers to the identifier of the channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ (i.e., the id of α from the description above); similarly, the value $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ refers to the identifier of the channel between $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ (i.e., β from the description above). The value $\gamma.\mathsf{fee} \in \mathbb{R}^{\geq 0}$ represents the fee charged by $\gamma.\mathsf{Ingrid}$ for her service of being an intermediary of γ. Finally, we introduce the attribute $\gamma.\mathsf{val} \in \mathbb{N} \cup \{\bot\}$. If $\gamma.\mathsf{val} \neq \bot$, then we call γ a *virtual channel with validity* and the value of $\gamma.\mathsf{val}$ represents the round number until which γ remains open. Channels with $\gamma.\mathsf{val} = \bot$ are called *virtual channels without validity*.

6.3.2 Security and efficiency goals

We briefly recall the properties of generalized channels as defined in [AEE+21] and state the additional properties that we require from virtual channels.

Security goals. Generalized ledger channels must satisfy three security properties, namely (S1) Consensus on creation, (S2) Consensus on update and (S3) Instant finality with punish. Intuitively, properties (S1) and (S2) guarantee that successful creation of a new channel as well as successful update of an existing channel happens if and only if both parties agree on the respective action. Property (S3) states that if a channel γ is successfully updated to the state $\gamma.\mathsf{st}$ and $\gamma.\mathsf{st}$ is the last state that the channel is updated to, then an honest party $P \in \gamma.\mathsf{users}$ can either enforce this state on the ledger or P can enforce a state where she gets all the coins locked in the channel. We say that a state st is *enforced* when a transaction with this state appears on the ledger.

Since virtual channels are generalized channels whose funding transaction is not posted on the ledger yet, the above-stated properties should hold for virtual channels as well with two subtle but important differences: (i) the creation of a virtual channel involves three parties (Alice, Ingrid, and Bob) and hence consensus on creation for virtual channels can only be fulfilled if all three parties agree on the creation; (ii) the finality (i.e., offloading) of the virtual channel depends on whether Alice is expected to offload the virtual channel within a predetermined validity period (virtual channel with validity VC-V) or the offload task is delegated to the intermediary Ingrid without having a predefined validity period (virtual channel without validity VC-NV). In order to account for these two differences, virtual channels should also satisfy the following properties:

(V1) Balance security: If γ is a virtual channel and $\gamma.\mathsf{Ingrid}$ is honest, she never loses coins, even if $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ collude.

(V2) Offload with punish: If γ is a virtual channel *without* validity (VC-NV), then $\gamma.\mathsf{Ingrid}$ can transform γ to a ledger channel. Party $P \in \gamma.\mathsf{users}$ can initiate the transformation which either completes or P can get financially compensated.

Table 6.1: Comparison of security and efficiency goals for ledger channels (L), virtual channels with validity (VC-V), and virtual channels without validity (VC-NV).

	L-Security	V-Security			Efficiency		
	S1 – S3	V1	V2	V3	E1	E2	E3
L	✓	-	-	-	✗	✓	✗
VC-V	✓	✓	✗	✓	✓	✓	✓
VC-NV	✓	✓	✓	✗	✓	✓	✓

(V3) Validity with punish: If γ is a virtual channel *with* validity (VC-V), then γ.Alice can transform γ to a ledger channel. If γ is not transformed into a ledger channel or closed before time γ.val, γ.Ingrid and γ.Bob can get financially compensated.

We first note that the instant finality with punish property (S3) does not provide any guarantees for Ingrid $\notin \gamma$.users, which is why we need to define (V1) for virtual channels. Properties (V2) and (V3) point out the main difference between VC-NV and VC-V. In a VC-NV γ, Ingrid is able to free her collateral from γ at any time by transforming the channel between Alice and Bob from a virtual channel to a ledger channel. Furthermore, in case Alice and Bob transform the virtual channel to a ledger channel or even misbehave, honest Ingrid is guaranteed that she will receive the collateral back. In a VC-V γ, Ingrid cannot transform a virtual channel into a ledger channel at any time she wants. Instead, there is a pre-agreed point in time, defined by γ.val, until when γ.users have to close the virtual channel or transform it into a ledger channel (Ingrid's collateral is freed in both cases). If γ.users fail to do so, Ingrid can get her collateral back through a punishment mechanism. Hence, γ.users have a guarantee that their VC-V will remain a virtual channel until a certain round, after which they must ensure its closure or transformation to avoid punishments.

Efficiency goals. Lastly, we define the following efficiency goals, which describe the number of rounds certain protocol steps require:

(E1) Constant round creation: Successful creation of a virtual channel takes a constant number of rounds.

(E2) Optimistic update: For a channel γ, this property guarantees that in the optimistic case when both parties in γ.users are honest, a channel update takes a constant number of rounds.

(E3) Optimistic closure: In the optimistic case when all parties in γ.users are honest, the closure of a virtual channel takes a constant number of rounds.

Let us stress that property (E2) is common for all off-chain channels (i.e., both ledger and virtual channels). The properties (E1) and (E3) capture the additional property of virtual channels that in the optimistic case when all parties behave honestly, the entire life-cycle of the channel is performed completely off-chain.

We compare the security and efficiency goals for different types of channels in Table 6.1. We formalize these properties as a UC ideal functionality in Appendix E.4.1.

6.3.3 Design Challenges for Constructing Virtual Channels

The main challenges that arise when constructing Bitcoin-compatible virtual channels stem from the need to ensure the security properties (V1) - (V3) as presented in the previous section. Namely, to guarantee balance security to the intermediary, we need

to ensure that the virtual channel creation and closure is reflected symmetrically and synchronously on both underlying ledger channels. We identify this as a challenge (C1). As we discuss in more detail below, this can be solved by giving the intermediary the right of a "last say" in the virtual channel creation and closure procedures. However, a malicious intermediary could abuse such power and block virtual channel closure indefinitely. Therefore, the second challenge (C2) is to design a punishment mechanism that allows virtual channel users to either enforce closure or claim financial compensation. We provide some further details below.

Synchronous create and close (C1). The creation and closure of a virtual channel are done by updating the underlying ledger channels. In order to guarantee balance security for the intermediary, we must ensure that updates on both ledger channels are symmetric and that either both of them succeed or both of them fail. That is, if the intermediary Ingrid loses coins in one ledger channel as a result of the virtual channel construction, then she has the guarantee of gaining the same amount of coins from the other ledger channel. Such an atomicity property can be achieved by allowing Ingrid to be the reacting party in both ledger channel update procedures. Namely, Ingrid has to receive symmetric update requests from both Alice and Bob before she confirms either of them.

As a result, Ingrid has the power to block a virtual channel creation and closure. For a virtual channel creation, this is not a problem. It simply represents the fact that Ingrid does not want to be an intermediary, and hence Alice and Bob have to find a different party. However, for virtual channel closing, this power of the intermediary results in a violation of the instant finality property for Alice and Bob, and requires a more involved mechanism.

Enforcing virtual channel state (C2). In contrast to standard ledger channels that rely on funding transactions that are published on the ledger, the funding transactions of a virtual channel are, in the optimistic case (i.e., when parties are honest), kept off-chain. In case of misbehavior (e.g., when malicious Ingrid refuses to close the virtual channel), however, honest parties must be able to publish the virtual channel funding transaction to the blockchain in order to enforce the latest state of the virtual channel. Unfortunately, the funding transactions can only be published if *both* of the underlying channels are closed in a state that funds the virtual channel. The fact that the virtual channel participants, Alice and Bob, respectively have control over just one of the underlying ledger channels further complicates this situation. For instance, one of the underlying ledger channels may be updated or closed maliciously at any time which would prevent the publishing of the funding transaction on the ledger.

6.3.4 Virtual Channel Protocol

We now show how to build virtual channels on top of generalized channels. We later discuss in Section 6.3.4 how our construction can be built over other channels such as Lightning and why generalized channels offer better efficiency.

As mentioned in the previous section, virtual channels are created and closed through an update of the underlying ledger channels. Hence, let us recall the update process of ledger channels, depicted as UpdateChan in Figures 6.4 and 6.5, before explaining our construction in more detail. The update procedure consists of 4 steps, namely (1) the *Initialization* step, during which parties agree on the new state of the channel, (2) the *Preparation* step, where parties generate the transactions with the given state, (3) the *Setup* during which parties exchange their application-dependent data (e.g., for building virtual channels), and finally (4) the *Completion* step where parties commit to the new state and revoke the old one. We refer the reader to [AEE+21] for more details.

High-level protocol description

We are now prepared to present a high-level description of our modular virtual channel protocol and explain how to solve the main technical challenges when designing virtual channels. In a nutshell, this modular protocol gives a generic framework on how to design virtual channels. Afterwards, we show how to instantiate this modular protocol with our virtual channel construction without validity. For the description of the instantiation with our construction with validity, we refer the reader to Appendix E.4.2. We present the formal pseudocode for the modular protocol as well as the instantiations with and without validity in Appendix E.4.3.

Create. Let γ be a virtual channel that $A := \gamma.\mathsf{Alice}$ and $B := \gamma.\mathsf{Bob}$ want to create, using their generalized ledger channels with $I := \gamma.\mathsf{Ingrid}$. At a high level, the creation procedure of a virtual channel is a synchronous update of the underlying ledger channels. Given the ledger channels, we proceed as follows (see Figure 6.4).

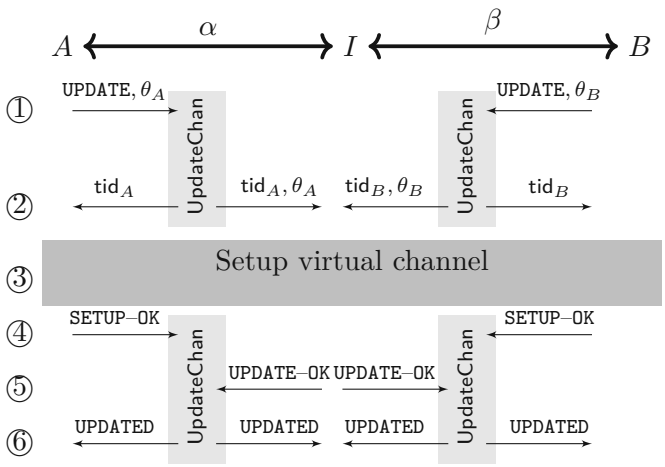

Figure 6.4: Modular creation procedure of a virtual channel on top of two ledger channels α and β.

As a first step, each party $P \in \{A, B\}$ initiates an update of the respective ledger channel with I (step ①) who, upon receiving both update requests, checks if the requested states

(i.e., θ_A and θ_B) are consistent. The parties use the identifiers tid_A and tid_B of their subchannels in order to build the virtual channel (step ②). Next, all three parties engage in a setup phase, in which the structure of the virtual channel is built (step ③). More concretely, all three parties agree on a funding transaction of the virtual channel which when published on the blockchain transforms the virtual channel to a ledger channel. When the setup phase is completed, i.e., the virtual channel structure has been built, the parties complete the ledger channel update procedures (step ④). It is crucial for the intermediary I to have the role of a reacting party during both channel updates. This gives her the power to wait until she is sure that both updates will complete successfully and only then give her the final update agreement (step ⑤). Upon successful execution, parties consider the channels as updated (step ⑥), which implies that the virtual channel γ was successfully created.

Update. Updating the virtual channel essentially works in the same way as the update procedure of a ledger channel. As long as the update is successful or peacefully rejected (meaning that the reacting party rejects the update), the parties act as instructed in the ledger channel protocol. The situation is more delicate when the update fails because one of the parties misbehaved and aborted the procedure.

We note that aborts during a channel update might cause a problematic asymmetry between the parties. For instance, when one party already signed the new state of the channel while the other one did not; or when one party already revoked the old state of the channel but the other one did not. In a standard ledger channel, these disputes are resolved by a force close procedure, meaning that the honest party publishes the latest valid state on the blockchain, thereby forcefully closing the channel. Hence, within a finite number of rounds, the dispute is resolved and the instant finality property is preserved. We apply a similar technique for virtual channels. The main difference is that a virtual channel is not funded on-chain. Hence, we first need to offload the virtual channel to the ledger. In other words, we first need to transform a virtual channel into a ledger channel by publishing its funding transaction on-chain. This process is discussed later in this section. Once the funding transaction is published, the dispute is handled in the same way as for ledger channels.

Close. The closure of a virtual channel is done by updating the underlying ledger channels α and β according to the latest state of the virtual channel $\gamma.\mathsf{st}$. To this end, each party $P \in \{A, B\}$ computes the new state for the ledger channel $\overrightarrow{\theta}_P := \{(c_P, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P}), (\gamma.\mathsf{cash} - c_P, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})\}$ where c_P is the latest balance of P in γ. All parties update their ledger channels according to this state.

In a bit more detail, the closing procedure of a virtual channel proceeds as follows (see Figure 6.5). Each party P initiates an update of the underlying ledger channel with state $\overrightarrow{\theta}_P$ (step ①). Since both ledger channels must be updated synchronously, I waits for both parties to initiate the update procedure. Upon receiving the states from both parties (step ②), I checks that the states are consistent and if so, she agrees to the update of

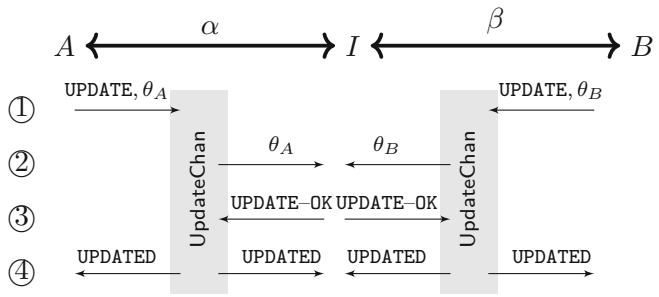

Figure 6.5: Modular close procedure of a virtual channel on top of two ledger channels α and β. For $P \in \{A, B\}$, $\overrightarrow{\theta}_P := \{(c_P, \texttt{One–Sig}_{\mathsf{pk}_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \texttt{One–Sig}_{\mathsf{pk}_I})\}$ where $\gamma.\mathsf{st} = \Big((c_P, \texttt{One–Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One–Sig}_{\mathsf{pk}_Q})\Big)$.

both ledger channels (step ③). Finally, after all parties have successfully revoked the previous ledger channel state, the virtual channel is considered to be closed.

In the pessimistic case (if the states $\overrightarrow{\theta}_A$ and $\overrightarrow{\theta}_B$ are inconsistent, revocation fails or I remains idle), parties must forcefully close their virtual channel by publishing the funding transaction (offloading) and closing the resulting ledger channel. This, together with the fact that I plays the role of the reacting party in its interactions with A and B, addresses the challenge (C1) as mentioned in Section 6.3.3.

Offload. During the offload procedure, parties try to publish the funding transaction of the virtual channel γ which effectively transforms the virtual channel into a ledger channel. In a nutshell, during this procedure, parties try to publish the commit and split transactions of both underlying ledger channels and afterward the funding transaction of the virtual channel. In case offloading is prevented by some form of malicious behavior, parties can engage in the punishment procedure to ensure that they do not lose any funds.

Punish. The concept of punishment in virtual channels is similar to that in ledger channels; namely in case that the latest state of a channel cannot be posted on the ledger, honest A or B are compensated by receiving all coins of the virtual channel while honest I will not lose coins. If the funding transaction of the virtual channel is posted on the ledger, the virtual channel is transformed into a ledger channel and parties can execute the regular punishment protocol for ledger channels. In addition to the ledger channel's punishment procedure, parties can punish if the funding transaction of γ cannot be published. Since this punishment, however, differs for each concrete instantiation, we will explain it in more detail for our protocol without validity in the following section (and in Appendix E.4.2 for the case with validity).

The offloading and punishment procedure together tackles challenge (C2) from Section 6.3.3.

Concrete Instantiation Without Validity

We now describe how the modular protocol explained above can be concretely instantiated with our construction for virtual channels without validity. **Create.** In our construction without validity, A and B must "prepare" the virtual channel during the setup procedure (step ③ in create of the modular protocol). This is done by executing the creation procedure of a regular ledger channel, i.e., they create a funding transaction with inputs tid_A and tid_B, as well as a commit and split transactions that spend the funding transaction. Once all three transactions are created, A and B sign them and exchange their signatures. Note that this corresponds to a normal channel opening, with the mere difference that the funding transaction is not published to the blockchain. In order to complete the virtual channel setup, A and B send the signed funding transaction to I who, upon receiving both signatures, sends her own signature on the transaction back to A and B. At this stage, the virtual channel is prepared, however, the creation is not completed yet. In order to finish the creation procedure, A, I, and B have to finish the update of their respective ledger channels. Once this is done, the virtual channel has been successfully created.

We illustrate the transaction structure prepared during the creation process in Figure 6.6. The funding transaction of the virtual channel tx^{f}, which is generated during the create procedure, takes as input coins from both, the ledger channel α (represented by $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$) and the ledger channel β (represented by $\mathtt{TX}_{\mathtt{s}}^{\mathtt{B}}$). Both ledger channels jointly contribute a total of $2c + f$ coins so that c coins are later used to setup the virtual channel and the remaining $c + f$ coins are I's collateral and the fees paid to I for providing the service for A and B.[1] I's collateral and fees in the funding transaction tx^{f} are the reason why I has to proactively monitor the virtual channel as she has an incentive to publish tx^{f} in case any party misbehaves.

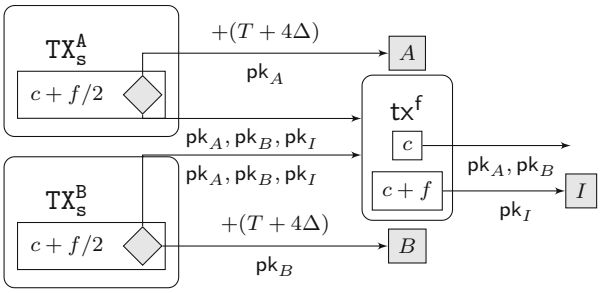

Figure 6.6: Funding of a virtual channel γ without validity. T upper bounds the number of off-chain communication rounds between two parties for any operation in the ledger channel.

Offload. I is always able to offload the virtual channel by herself (i.e., without having to cooperate with another party) which guarantees that I can redeem her collateral at

[1] For simplicity we assume each of the parties contributes $f/2$ coins to I's total fees in addition to $c/2$ coins for funding the virtual channel.

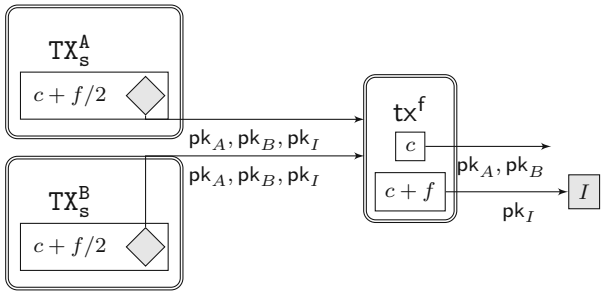

Figure 6.7: Transactions published after a successful offload.

any time. We note that $P \in \{A, B\}$ can also initiate the offloading by publishing the commit and split transaction of their respective ledger channels. This forces I to publish the commit and split transactions of the respective other ledger channel since I loses her collateral to P otherwise.

More precisely, if I wishes to offload the virtual channel γ and retrieve her collateral and fees, she can close both of her ledger channels with A and B (i.e., α and β) and publish the funding transaction of the virtual channel i.e., tx_f. This is possible as I is part of both ledger channels. A or B, on the other hand, are respectively part of only one ledger channel and hence they cannot offload the virtual channel individually. However, they can force I to offload by publishing the commit and split transactions of their respective channel with I (we will elaborate on this in the description of the punishment mechanism). Figure 6.7 illustrates the transactions that are posted on the blockchain in case of a successful offload. The figure shows that the split transactions of both underlying ledger channels have to be published such that eventually the funding transaction of the virtual channel can be published which completes the offloading procedure.

Punish. Party $P \in \{A, B\}$ can punish I by taking all the coins on their respective ledger channels if the funding transaction of the virtual channel γ is not published on the ledger. In other words, it is I's responsibility to ensure that the state of her ledger channels with A and B are not updated while γ is open. Furthermore, upon one of the subchannels being closed, I must close the other subchannel in order to guarantee that both parties can post tx^f.

Let us now get into more details. Assume that A's ledger channel with I is closed, but the funding transaction tx^f cannot be published on the blockchain. This means that I's channel with B (i.e., β) is still open or has been closed in a different state such that tx^f cannot be published. In other words, Ingrid acted maliciously by wrongfully closing β in a different state or by not closing β at all. In this case, A must be able to get all the coins from her channel with Ingrid. This punishment works as follows: After A publishing the split transaction of α, I is given a certain time period to close her channel with B and publish the virtual channel's funding transaction tx^f. If I fails to do so in the prescribed time period, A receives all coins in her channel with I.

We note that in this scenario, B (instead of I) might have been the malicious party by

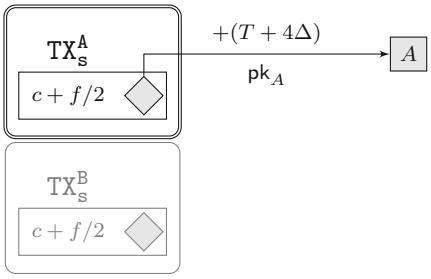

Figure 6.8: Transactions published after A successfully executed the punishment procedure. The grayed transaction $\mathtt{TX}_{\mathtt{s}}^{\mathtt{B}}$ indicates that this transaction has not been published.

closing β in an outdated state, thereby leaving I no option to publish tx^{f}. However, in this case, I can punish B via the punishment mechanism of the underlying ledger channel and earn all the coins in β. Therefore, I will remain financially neutral as she gets punished by A but simultaneously compensated by B. Figure 6.8 illustrates the transactions that are posted on the blockchain in the case of A successfully executing the punishment mechanism. The case where B executes the punishment mechanism is analogous.

Further discussion regarding our constructions

In the following, we present further considerations regarding our protocol, including remarks on concurrency, a discussion on how the protocol can be built on top of Lightning channels, and a brief description of our virtual channel construction with validity that we detail in Appendix E.4.2.

Concurrency. When creating a virtual channel, we need to lock the underlying ledger channels α and β (i.e., no further updates can be made on the ledger channels as long as the virtual channel is open). This, however, is undesirable, because in most cases the ledger channels will have more coins available than what is needed for funding the virtual channel. We emphasize that this issue can be easily addressed (and hence supporting full concurrency) by using the channel splitting technique discussed in [AEE+21]. This means that before constructing the virtual channel Alice-Bob, parties would first *split* each underlying ledger channel off-chain in two channels: (i) one would contain the exact amount of coins for the virtual channel and (ii) the other one would contain the remaining coins that can be used in the underlying ledger channel.

Virtual channels over Lightning. We will now discuss how our virtual channel constructions can be built on top of any ledger channel infrastructure that uses a *revocation/punishment* mechanism such as the Lightning Network [PD16]. The main complication arises from the fact that ledger channel constructions other than generalized channels require two commit transactions per channel state (one for each party). As depicted in Figure 6.9 (and unlike generalized channels in Figure 6.2), Alice and Bob each have a commit transaction $\mathtt{TX}_{\mathtt{c}}^{\mathtt{A}}$ and $\mathtt{TX}_{\mathtt{c}}^{\mathtt{B}}$ which spends the funding transaction tx^{f}

and distributes the coins. Therefore, in such channel constructions, it is a priori unclear which of these commit transactions will be posted and accepted on the blockchain (note that only one of them can be successfully published) and hence building applications (e.g., virtual channels) on top of such ledger channels becomes complex.

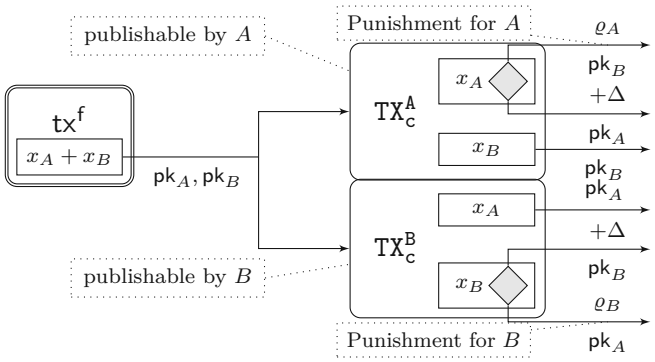

Figure 6.9: A Lightning style payment channel where A has x_A coins and B has x_B coins. Δ upper bounds the time needed to publish a transaction on a blockchain. condition ϱ_A represents the verification of A' revocation secret and h represents the verification of B' revocation secret.

In more detail, assume Alice and Bob want to build a virtual channel γ on top of their respective Lightning ledger channels with Ingrid, where both ledger channels consist of two commit transactions respectively (i.e., $(\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{IA}})$ for the channel between Alice and Ingrid and $(\mathtt{TX}_{\mathsf{c}}^{\mathtt{B}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{IB}})$ for the channel between Bob and Ingrid). All three parties now have to make sure that the virtual channel can be funded (i.e., that the funding transaction of γ can be published to the blockchain) even in case of malicious behavior. To ensure this, parties have to prepare the funding transaction of γ with respect to all possible combinations of the commit transactions of the respective underlying ledger channels. Since there are four such combinations ($(\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{B}})$, $(\mathtt{TX}_{\mathsf{c}}^{\mathtt{A}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{IB}})$, $(\mathtt{TX}_{\mathsf{c}}^{\mathtt{IA}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{B}})$ and $(\mathtt{TX}_{\mathsf{c}}^{\mathtt{IA}}, \mathtt{TX}_{\mathsf{c}}^{\mathtt{IB}})$), parties have to prepare four funding transactions for γ. Hence, updating such a virtual channel requires repeating the update procedure for all four funding transactions.

As generalized channels require only a single commit transaction per channel state building virtual channels on top of generalized channels offers a significant efficiency improvement in terms of off-chain communication complexity (see Section 6.5 for the detailed comparison).

Virtual Channels With Validity. Note that so far we described our protocol without validity where the virtual channel can be offloaded by the intermediary whenever she wants. The drawback of this construction is that Ingrid needs to be proactive during the lifetime of the virtual channel, i.e., she has to constantly monitor the channel for potential misbehavior of Alice or Bob. This might be undesirable in scenarios where Ingrid plays the role of the intermediary in not just one but many different virtual channels at the

same time (e.g., if Ingrid is a channel hub). For this reason, we developed an alternative solution which we call virtual channels with validity. In this solution, each virtual channel has a predetermined time (which we call validity) which indicates until when the channel has to be closed again. If the channel is still open after this time, Ingrid has to become proactive in order to receive her collateral back. The obvious advantage of this approach is that Ingrid can remain inactive until the validity of a channel expires. The details of this protocol can be found in Appendix E.4.2.

6.4 Security Model and Analysis

In order to model and prove the security of our virtual channel protocols, we use the global UC framework (GUC) [CDPW07] as in [AEE+21]. This framework allows for a global setup which we utilize to model a public blockchain. More precisely, our protocol uses a global ledger functionality $\mathcal{L}(\Delta, \Sigma)$, where Δ upper bounds the blockchain delay, i.e., the maximum number of rounds required to publish a transaction, and Σ is the signature scheme used by the blockchain. In this section, we only give a high-level idea behind our security analysis in the UC framework and refer the reader to Appendices E.1, E.3.1, E.3.3 and E.4.1 for more details.

As a first step, we define the expected behavior of a virtual channel protocol in the form of an *ideal functionality* $\mathcal{F}_V$. The functionality defines the input/output behavior of a protocol, its impact on the global setup (e.g., ledger), and the possible ways an adversary can influence its execution (e.g., delaying messages). In order to prove that a concrete protocol is a secure virtual channel protocol, one must show that the protocol *emulates* the ideal functionality $\mathcal{F}_V$. This means that any attack that can be mounted on the protocol can also be mounted on the ideal functionality, hence the protocol is at least as secure as the ideal specification given by $\mathcal{F}_V$.

The proof of emulation consists of two steps. First, one must design a *simulator*, which simulates the actions of an adversary on the real-world protocol by interacting with the ideal functionality. Second, it must be shown that the execution of the real-world protocol being attacked by a real-world adversary is indistinguishable from the execution of the ideal functionality communicating with the constructed simulator. In UC, the ppt distinguisher who tries to distinguish these two executions is called the *environment*.

The main challenge when designing a simulator is to make sure that the environment sees transactions being posted on the ledger in the same round in both worlds. In addition, our simulator needs to ensure that the ideal functionality outputs the same set of messages in the same round as the protocol. We reduce the indistinguishability of the two executions to the security of the cryptographic primitives used in our protocol.

One of the advantages of using UC is its composability. In other words, one can use an ideal functionality in a black-box way in other protocols. This simplifies the process of designing new protocols as it allows to reuse existing results and enables modular protocol designs. We utilize this nice property of the UC framework and use the ideal functionality of the generalized channel from [AEE+21] when designing our virtual channel protocol.

We only mention the main security theorem here and provide a high-level proof sketch here. We refer the reader to Appendix E.5.1 for the full proof.

Theorem 6. *Let Σ be a signature scheme that is strongly unforgeable against chosen message attacks. Then for any ledger delay $\Delta \in \mathbb{N}$, the virtual channel protocol without validity as described in Section 6.3.4 working in $\mathcal{F}_{preL}(3,1)$-hybrid, UC-realizes the ideal functionality $\mathcal{F}_V(3)$.*

We now give a proof sketch to show that the two properties (V1) Balance security and (V2) Offload with punish hold for honest parties. To this end, we analyze all possible cases in which the underlying ledger channels are maliciously closed, i.e., the cases when the virtual channel cannot be offloaded anymore. Note that if the virtual channel is offloaded, it is effectively transformed into a generalized ledger channel and satisfies the security properties of generalized channels.

If all parties behave honestly (V1) and (V2) hold trivially as I is always able to offload the virtual channel by publishing all transactions $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$, $\mathtt{TX}_{\mathtt{s}}^{\mathtt{B}}$ and $\mathtt{tx}^{\mathtt{f}}$. Furthermore, neither A nor B would ever lose their coins. Now consider the case where one of the underlying channels, e.g., the channel between B and I is closed in a different state such that $\mathtt{tx}^{\mathtt{f}}$ cannot be posted on the blockchain anymore (the case for the channel between A and I is analogous). As an honest A would not update her channel with I as long as the virtual channel is open, there are only two possible situations: (i) A is able to post $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ which allows her to punish I (see Figure 6.8), or (ii) I has maliciously closed her channel with A in an outdated and revoked state. In this case, A is able to punish I according to property (S3), i.e., instant finality with punish, of the underlying ledger channel (see Section 6.2 and Figure 6.2 for more details on the punishment of the underlying channel). Therefore, (V2) is satisfied for A, since she can punish I and get financially compensated. Now let us analyze the maliciously closed channel between B and I, let us denote it β. If both parties are malicious, we do not need to prove anything as (V1) and (V2) should only hold for honest parties. In case B is honest, I must have closed β in an old state which would allow B to punish I. Hence (V2) holds and we do not need to prove (V1) as I is malicious. Analogously, if I is honest, malicious B must have closed β in an old state, and hence I can punish B. Hence (V1) holds and we do not need to prove (V2) for malicious B). Hence, (V1) and (V2) hold for all honest parties.

6.5 Performance evaluation

In this section, we first study the storage overhead on the blockchain as well as the communication overhead between users using virtual channels. For each of these aspects, we evaluate both constructions (i.e., with and without validity) built on top of both generalized channels as well as Lightning channels and compare them. Finally, we evaluate the advantages of virtual channels over ledger channels in terms of routing communication overhead and fee costs. As testbed [Bit20], the transactions are created in Python using the library `python-bitcoin-utils` and the Bitcoin *Script* language. To showcase

compatibility and feasibility, we deployed these transactions successfully on the Bitcoin testnet.

6.5.1 Communication overhead

We analyze the communication overhead imposed by the different operations, such as `CREATE`, `UPDATE`, `OFFLOAD`, and `CLOSE`, by measuring the byte size of the transactions that need to be exchanged as well as the cost in USD necessary for posting the transactions that need to be published on-chain. The cost in USD is calculated by taking the price of 18803 USD per Bitcoin, and the average transaction fee of 104 satoshis per byte all of them at the time of writing. We detail in Table 6.2 the aforementioned costs measured for both virtual channel constructions building on top of generalized channels and on top of Lightning channels.

Table 6.2: Evaluation of the virtual channels. For each operation, we show the number of on-chain and off-chain transactions (*# txs*) and their *size* in bytes. For on-chain transactions, *cost* is in USD and estimates cost of publishing them on the ledger.

Operations	Generalized Channels										Lightning Channels									
	VC-NV					VC-V					VC-NV					VC-V				
	on-chain			off-chain		on-chain			off-chain		on-chain			off-chain		on-chain			off-chain	
	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size
`CREATE`	0	0	0	7	2829	0	0	0	8	2803	0	0	0	16	7704	0	0	0	14	5722
`UPDATE`	0	0	0	2	695	0	0	0	2	695	0	0	0	8	2824	0	0	0	4	1412
`OFFLOAD`	5	2134	41.73	0	0	6	2108	41.22	0	0	3	1800	35.20	0	0	4	1778	34.77	0	0
`CLOSE` (opt)	0	0	0	4	1390	0	0	0	4	1390	0	0	0	4	1412	0	0	0	4	1412
`CLOSE` (pess)	7	2829	55.32	0	0	8	2803	54.81	0	0	4	2153	42.10	0	0	5	2131	41.67	0	0

Perhaps the most relevant difference to ledger channels in practice is, in the `CREATE` and the optimistic `CLOSE` case, we do not have any on-chain transactions. This implies no on-chain fees for the opening and closing of virtual channels.

Virtual channels over generalized channels. For the creation of a virtual channel (`CREATE` operation) on top of generalized channels, we need to update both ledger channels to a new state that can fund the virtual channel, requiring to exchange $2 \cdot 2$ transactions with 1494 (VC-NV) or 1422 (VC-V) bytes. Additionally, we need 640 bytes for tx^{f} (VC-NV) or 309 + 377 bytes for tx^{f} and $\mathsf{tx}_{\mathsf{refund}}$ (VC-V). Finally, for both VC-NV and VC-V, we need the transactions representing the state of the virtual channel itself which requires 431 bytes for $\mathtt{TX}_{\mathsf{c}}$ and 264 bytes for $\mathtt{TX}_{\mathsf{s}}$. This complete process results in 7 (VC-NV) or 8 (VC-V) transactions with a total of 2829 (VC-NV) or 2803 (VC-V) bytes. Forcefully closing (`CLOSE`(pess) operation) and offloading (`OFFLOAD` operation) requires the same set of transactions as with `CREATE`, minus the commitment and the split transaction (695 bytes) of the virtual channel in the latter case, both on-chain. Finally, we observe that the `UPDATE` and the optimistic `CLOSE`(opt) operation require 2 transactions (695 bytes) for both constructions, as they are designed as an update of a ledger channel.

Virtual channels over Lightning channels. Building virtual channels on top of Lightning channels yields the following results. Instead of one commitment and one split transaction per ledger channel, we now need two commitment transactions per ledger channel, each of size 580 (VC-NV) or 546 (VC-V) bytes. Due to the fact that in both

ledger channels, either commitment transaction can be published, we now need four tx^{f} of 640 bytes each (VC-NV) or two tx^{f} of 309 and four $\mathsf{tx}_{\mathsf{refund}}$ of 377 bytes (VC-V). For every tx^{f}, we need two commitment transactions of 353 bytes (in total, $8 \cdot 353$ in VC-NV or $4 \cdot 353$ in VC-V). For `OFFLOAD`, only one commitment transaction per ledger channel needs to be published, along with one tx^{f} (for VC-NV) and tx^{f} plus $\mathsf{tx}_{\mathsf{refund}}$ (for VC-V). `CLOSE`(pess), needs to publish a commitment transaction in addition to `OFFLOAD`, resulting in 2153 (VC-NV) or 2131 (VC-V) bytes.

6.5.2 Comparison to payment channel networks

In this section, we compare virtual channels to multi-hop payments in a payment channel network (PCN). In a PCN, users route their payments via intermediaries. During the routing of a transaction tx, each intermediary party locks $\mathsf{tx.cash}$ coins as a "promise to pay" in their channels, a payment commitment that can technically be implemented as a Hash-Time Lock Contract (HTLC), e.g. as in the Lightning Network [PD16]. We now evaluate the difference in communication overhead and fee costs compared to virtual channels, summarize them in Table 6.3, and illustrate them in Figure 6.10.

Routing communication overhead. When performing a payment between Alice and Bob via an intermediary Ingrid in a multi-hop payment over generalized channels, the participants need to update both generalized channels with a "promise to pay", which require 2 transactions or 818 bytes per channel when implemented as HTLC. If they are successful, both generalized channels need to be updated again to "confirm the payment" (again, 2 transactions or 695 bytes per channel). This whole process results in 8 transactions or $2 \cdot 818 + 2 \cdot 695 = 3026$ off-chain bytes that need to be exchanged. Generically, if the parties want to perform n sequential payments, they need to exchange $8 \cdot n$ transaction with a total of $3026 \cdot n$ bytes.

Assume now that Alice and Bob were to perform the payment over a virtual channel without validity instead and that this virtual channel is not yet created. As shown in Table 6.2, they need to open the virtual channel for 2829 bytes, where they set the balance of the virtual channel already to the correct state after the payment, and then close it again for 1390 bytes, resulting in a total of 4219 off-chain bytes. However, if we again consider n sequential payments, the result would be $9 + 2 \cdot n$ transactions or $3524 + 695 \cdot n$ bytes, which supposes a reduction of $2331 \cdot n - 3524$ bytes with respect to relying on generalized channels only. This means that a virtual channel is already cheaper if only two (or more) sequential transactions are performed. We obtain similar results if we consider virtual channels with validity instead. For Lightning channels, the overhead is larger for both the multi-hop payment and the VC setting (Table 6.3).

Fee costs. In a multi-hop payment tx in a PCN, the intermediary user Ingrid charges a base fee (BF) for being online and offering the routing service and relative fee (FR) for locking the amounts of coins ($\mathsf{tx.cash}$) and changing the balance in the channel, so that $\mathsf{fee}(\mathsf{tx}) := \mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$. Note that at the time of writing, the fees are $\mathsf{BF} = 1$ satoshi and $\mathsf{FR} = 0.000001$.

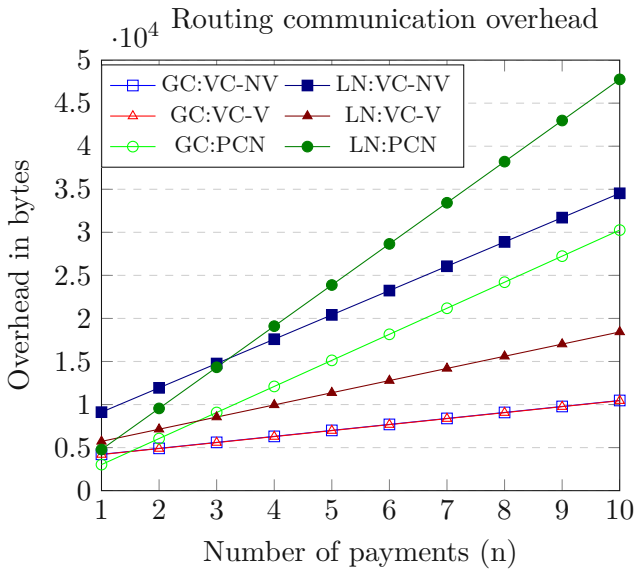

Figure 6.10: Pictorial illustration of Table 6.3.

Table 6.3: Comparison of virtual channels (VC) to multi-hop payments (PCN) showing the overhead in bytes for a different number of payments and the difference in fees.

	Overhead in bytes			fees
	1 paym.	2 paym.	n payments	tx.cash in n payments
GC: PCN	3026	6052	$3026 \cdot n$	$\mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$
GC: VC-NV	4219	4914	$3524 + 695 \cdot n$	$\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$
GC: VC-V	4193	4888	$3498 + 695 \cdot n$	
LN: PCN	4776	9552	$4776 \cdot n$	$\mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$
LN: VC-NV	9116	11940	$6292 + 2824 \cdot n$	$\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$
LN: VC-V	5722	7134	$4310 + 1412 \cdot n$	

In a virtual channel setting, γ.Ingrid can charge a base fee to collaborate to open and close the virtual channel, and also a relative fee to lock collateral coins in the virtual channel. However, no fees per payment are charged by Ingrid as she does not participate in them (and even does not know how many end-users performed)1. Let us now investigate the case of paying tx.cash in n micropayments of equal value. In PCN case, the total cost would be $\sum_{i=1}^{n} \mathsf{BF} + \mathsf{FR} \cdot \frac{\mathsf{tx.cash}}{n} = \mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$. Whereas, in the virtual case, the parties first create a virtual channel γ with balance tx.cash, and they will handle the micropayments in γ. Thereby, the cost would be only the opening cost of the virtual channel, for which we assumed $\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$. Thus, if Alice and Bob would make more than one transaction, i.e., $n > 1$, it is beneficial to use virtual channels for reducing the fee costs by $\mathsf{BF} \cdot (n - 1)$.

Summary. We find that the best construction in practice is the combination of virtual channels on top of generalized channels, as this yields the least overhead after only two or more sequential payments. However, building virtual channels over LN channels also

yields less overhead than multi-hop PCN payments over LN.

6.6 Related Work

In this section, we position this work in the landscape of the literature for off-chain payment protocols.

Payment Channels. Starting from the Lightning Channels construction [PD16], the idea of 2-party payment channels has been largely used in academia and industry as a building block for more complex off-chain payment protocols. More recently, Aumayr et al. [AEE+21] have proposed a novel construction for 2-party payment channels that overcome some of the drawbacks of the original Lightning channels. While their benefit in terms of scalability is out of any doubt by now, payment channels are limited to payments between two users and consequently its overall utility.

A concurrent work [JLT20] has also proposed a virtual channel construction over Bitcoin. However, their construction uses decreasing time-locks instead of a punishment mechanism in order to guarantee that only the latest state can be posted on the blockchain. As a consequence, their construction only allows a fixed number of transactions to be made during the lifetime of the virtual channel. This is quite restrictive as it requires users to close and open new virtual channels more frequently which goes against the purpose of virtual channels. Note that one cannot simply increase the time-lock as this would essentially lock the coins of the users for a longer period of time. Furthermore, our constructions are *generalized* virtual channels, i.e., they are not limited to just payments, but rather allow to run any Bitcoin script off-chain. In addition, we propose a modular approach compared to the monolithic construction in [JLT20]. Finally, our work proposes two protocols, which each have their advantages in different use cases.

Payment Channel Networks (PCN) and Payment Channel Hub (PCH). A PCN allows a payment between two users that do not share a payment channel but are however connected through a path of payment channels. The notion of PCN started with the deployment of Lightning Network [PD16] for Bitcoin and Raiden Network [Rai17] for Ethereum and has been widely studied in academia to research into different aspects such as privacy [MMSK+17, MMS+19], routing of payments [RMKG18], collateral management [EMSM19] and others. Similar to PCN, different constructions for PCH exist [TMSM21a, HAB+17, BCG+14] that allow a payment between two users through a single intermediary, the payment hub. PCNs and PCHs, however, share the drawback that each payment between two users requires the active involvement of the intermediary (or several intermediaries in the case of PCH), which reduces the reliability (e.g., the intermediary can go offline) and increases the cost of the payment (e.g., each intermediary charges a fee for the payment).

State Channels. Several works [DEF+19b, DFH18, MBB+19b, CCF+21] have shown how to leverage the highly expressive scripting language available at Ethereum to construct (multi-party) state channels. A state channel allows the involved parties to carry out off-

chain computations, possibly other than payments. Closer to our work, Dziembowski et al. [DEFM19] showed how to construct a virtual channel leveraging two payment channels defined in Ethereum. These approaches are, however, highly tight to the functionality provided by the Ethereum scripting language and their constructions cannot be reused in other cryptocurrencies. In this work, we instead show that virtual channels can be constructed from digital signatures and timelock mechanism only, which makes virtual channels accessible for virtually any cryptocurrency system available today.

6.7 Conclusion

Current PCNs route payments between two users through intermediate nodes, making the system less reliable (intermediaries might be offline), expensive (intermediaries charge a fee per payment), and privacy-invasive (intermediate nodes observe every payment they route). To mitigate this, recent work has introduced the concept of virtual channels, which involve intermediaries only in the creation of a bridge between payer and payee, who can later on independently perform arbitrarily many off-chain transactions. Unfortunately, existing constructions are only available for Ethereum, as they rely on its account model and Turing-complete scripting language.

In this work, we present the first virtual channel constructions that are built on the UTXO model and require a scripting language supported by virtually every cryptocurrency, including Bitcoin. Our two protocols provide a tradeoff on who can offload the virtual channel, similar to the preemptible vs. non-preemptible virtual machines in the cloud setting. In other words, our virtual channel construction without validity is more suitable for intermediaries who can monitor the blockchain regularly, such as payment channel hubs, but can also close the virtual channel at any time if desired. Our virtual channel protocol with validity however, is more suitable for light intermediaries who do not wish to be active during the lifetime of the virtual channel but cannot close the virtual channel before its validity has expired. We formalize the security properties of virtual channels in the UC framework, proving that our protocols constitute a secure realization thereof. We have prototyped our protocols and evaluated their efficiency: for n sequential payments in the optimistic case, they require $9+2\cdot n$ off-chain transactions for a total of $3524+695\cdot n$ bytes, with no on-chain footprint.

As mentioned in the introduction of this work, the task of designing secure virtual channels has been proven to be challenging even on a cryptocurrency like Ethereum [DEFM19] which supports smart contract execution. Unsurprisingly, this task becomes even more complex when building virtual channels for blockchains that support only a limited scripting language as it is not possible to take advantage of the full computation power of Turing complete smart contracts. Due to these significantly differing underlying assumptions (smart contracts vs. limited scripting languages), the virtual channel protocols based on Ethereum [DEFM19] and the protocols presented in this work are incomparable. We emphasize that we view our virtual channel constructions as complementary to the one presented in [DEFM19], as we do not aim to improve the construction of [DEFM19] but rather extend the concept of virtual channels to a broader class of blockchains.

We conjecture that it is possible to recursively build virtual channels on top of any two underlying channels (either ledger, virtual, or a combination of them), requiring to adjust the timings for offloading channels: users of a virtual channel at layer k should have enough time to offload the (virtual/ledger) channels at layers 1 to $k-1$. Additionally, we envision that while virtual channels without validity might serve as a building block at any layer of recursion, virtual channels with validity period may be more suitable for the top layer as they have a predefined expiration time after which they would require to offload in any case all underlying layers. We plan to explore the recursive building of virtual channels in the near future. Additionally, we conjecture that virtual channels help with privacy, but we leave a formalization of this claim as interesting future work, as it involves a quantitative analysis that falls off the scope of this work.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program *FA 1320/1-1*, by the *DFG CRC 1119 CROSSING* (project S7), by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC), by the Austrian Science Fund (FWF) through PROFET (grant agreement P31621) and the Meitner program (grant agreement M 2608-G27), by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant agreement 13808694) and the COMET K1 projects SBA and ABC, by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP), by CoBloX Labs and by the ERC Project PREP-CRYPTO 724307.

CHAPTER 7

Breaking and Fixing Virtual Channels: Domino Attack and Donner

Abstract

Payment channel networks (PCNs) mitigate the scalability issues of current decentralized cryptocurrencies. They allow for arbitrarily many payments between users connected through a path of intermediate payment channels, while requiring interacting with the blockchain only to open and close the channels. Unfortunately, PCNs are (i) tailored to payments, excluding more complex smart contract functionalities, such as the oracle-enabling Discreet Log Contracts and (ii) their need for active participation from intermediaries may make payments unreliable, slower, expensive, and privacy-invasive. Virtual channels are among the most promising techniques to mitigate these issues, allowing two endpoints of a path to create a direct channel over the intermediaries without any interaction with the blockchain. After such a virtual channel is constructed, (i) the endpoints can use this direct channel for applications other than payments and (ii) the intermediaries are no longer involved in updates.

In this work, we first introduce the Domino attack, a new DoS/griefing style attack that leverages virtual channels to destruct the PCN itself and is inherent to the design adopted by the existing Bitcoin-compatible virtual channels. We then demonstrate its severity by a quantitative analysis on a snapshot of the Lightning Network (LN), the most widely deployed PCN at present. We finally discuss other serious drawbacks of existing virtual channel designs, such as the support for only a single intermediary, a latency and blockchain overhead linear in the path length, or a non-constant storage overhead per user.

We then present Donner, the first virtual channel construction that overcomes the shortcomings above, by relying on a novel design paradigm. We formally define and prove security and privacy properties in the Universal Composability framework. Our evaluation shows that Donner is efficient, reduces the on-chain number of transactions for disputes from linear in the path length to a single one, which is the key to preventing Domino attacks, and reduces the storage overhead from logarithmic in the path length to constant. Donner is Bitcoin-compatible and can be easily integrated into the LN.

This chapter presents the results of a collaboration with Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei, which was published at the Network and Distributed System Security Symposium (NDSS) in 2023 under the title "Breaking and Fixing Virtual Channels: Domino Attack and Donner". I am the main author of this paper. I am responsible for the idea, discovery, and write-up of the Domino attack, analyzing the impact of the Domino attack, designing and writing the protocol, formalization in the UC framework, defining the security and privacy properties, writing the proofs, writing a proof-of-concept evaluation, conducting the experiments and comparison to related work. Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei were the general advisors and contributed with continuous feedback.

7.1 Introduction

Payment channels (PCs) have emerged as one of the most promising solutions to the limited transaction throughput of permissionless blockchains, with the Lightning Network [PD16] being the most popular realization thereof in Bitcoin. A PC enables arbitrarily many payments between two users while requiring to commit only two transactions to the ledger: one to open and another to close the channel. Aside from payments, several applications proposed so far benefit from the scalability gains of 2-party PCs [Dry17, BK14, BDW17]. Recent work [AEE+21] has further shown how to lift any operation supported by the underlying blockchain to the off-chain setting, thereby further expanding the class of supported off-chain applications.

Creating PCs between all pairs of users (i.e., a clique) is economically infeasible, as users must lock coins for each PC and funding occurs on-chain. On-demand creation of PCs with any potential partner is also infeasible due to the need for on-chain transactions for opening and closing each channel, which results in on-chain fees, long confirmation times (around 1h in Bitcoin) and again impacts the blockchain throughput. As a result, single PCs are instead linked together to form PCNs, using paths of PCs to connect two users instead of opening a PC between them. The interactions of PCN users can be classified into synchronization protocols and virtual channels.

Synchronization protocols. Synchronization protocols, e.g., [EMSM19, AMSKM21, MBB+19b, MMS+19, MMSK+17, PD16], allow a sender to pay a receiver when they are connected by a path of PCs, atomically updating the balance of all PCs along the path. Although some of these synchronization protocols are deployed in practice (e.g., for multi-hop payments in the Lightning Network), there are several drawbacks:

(i, *online assumption*) they require users in the path to be online; (ii, *reliability*) each intermediate user must participate, making the payment less reliable; (iii, *cost*) each intermediate charges a fee per synchronization round; (iv, *latency*) the latency of the application increases along with the number of intermediaries (e.g., in the Lightning Network up to one-day latency per channel); (v, *privacy*) intermediaries are aware of every single operation; and (vi, *efficiency*) they can handle only a limited number of simultaneous payments (e.g., 483 in the Lightning Network) [dev]. Finally, and perhaps more importantly, current synchronization protocols are tailored to payments. Supporting 2-party applications (as the ones mentioned before) would require thus to come up with a synchronization protocol for each application. Apart from being a burden, it is not trivial to design such protocols tailored to applications beyond payments, as exemplified by the recent quest in the Bitcoin community about the realization of Discreet Log Contracts across multiple hops [DLC21].

Virtual channels. Virtual channels (VC) [DEFM19, DFH18, DEF+19b, Per20, AME+21, JLT20, KL] allow two users connected by a path of PCs to establish a direct connection, bypassing intermediaries. Intuitively, a VC is akin to a PC, but instead of being opened by an on-chain transaction, it is opened off-chain using funds from the path of PCs. Therefore, the opening phase involves all intermediaries, besides the endpoints. Once established, however, updates can proceed without the involvement of any intermediaries. In this manner, VCs overcome the aforementioned drawbacks of synchronization protocols: (i) intermediaries are no longer required to be online; (ii) the reliability of the channel does not depend on intermediaries; (iii) intermediaries do not charge a fee for each usage of the channel (perhaps only once to create and close the VC); (iv) the latency does not depend on intermediaries; (v) intermediaries do not learn each single VC update; (vi) a PC can host several VCs, each of which can be used to dispense up to 483 payments or potentially more VCs, bypassing the limitation on the number of payments in PCNs.

Since VCs can be used just as PCs, they constitute the most promising solution to perform repeated transactions as well as applications different from payments (e.g., [Dry17, BK14, BDW17]) between any pair of users connected by a path of PCs. In fact, applications built on top of PCs can be smoothly lifted to VCs, which constitutes a crucial improvement over synchronization protocols.[1] For instance, VCs support Discreet Log Contracts [Dry17], an application that has received increased attention lately and that intuitively allows for bets based on attestations from an oracle on real-world events. As compared to PCs, VCs offer the same advantages while requiring no on-chain transaction for their setup, thereby dispensing from the associated blockchain delays, on-chain fees, and on-chain footprints. This makes it possible to keep VCs short-lived, to frequently close, open, or extend them based on current needs. For a more detailed discussion see Appendix F.1.

VC constructions are difficult to design since the balance of honest parties needs to be ensured even in the presence of malicious, and possibly colluding intermediaries/endpoints.

[1] VCs expose all the functionalities of a PC and can be used interchangeably as a building block for off-chain applications, see Section 7.5.

The first constructions have been proposed for blockchains supporting Turing-complete scripting languages based on the account model, like Ethereum [DEFM19, DFH18, DEF+19b]. In such blockchains, VC constructions are somewhat easier to design: For instance, stateful smart contracts can resolve conflicts on the current state of VCs by associating a different version number to each state update and, in case of conflict, by selecting the highest number as the valid state. Indeed, Ethereum-based constructions are based on this idea and do not suffer from the Domino attack presented in this paper. Unfortunately, this reliance on Turing-complete scripting languages makes these constructions incompatible with many of the cryptocurrencies available today, including Bitcoin itself.

It is not only practically relevant but also theoretically interesting to investigate what is the minimum scripting functionalities necessary to design secure VCs. Therefore, a bit later VC constructions have been proposed also for blockchains with a less expressive scripting language and based on the Unspent Transaction Output (UTXO) model (i.e., Bitcoin-compatible) [AME+21, JLT20, KL]. Throughout the rest of this paper, we investigate VCs built on these blockchains if not specified otherwise. All of these VC constructions share one common design pattern: The VC is funded from all underlying PCs. We refer to this design pattern as *rooted VCs* and illustrate it on a high level in Figure 7.1(a.1). Because VCs are, unlike PCs, not funded on-chain, they rely on an operation called *offloading*, which transforms a VC to a PC. This is important for honest users so they can enforce their balance in case the other user misbehaves: first transforming the VC to a PC by putting the VC funding on-chain, and second using the means provided by the PC to enforce their balance. Rooted designs enable both endpoints to offload the VC, but because they are funded by all underlying PCs, every underlying PC has to be closed on-chain (see Figure 7.1(a.2)).

Table 7.1: Comparison to other multi-hop VC protocols. * by synchronizing all channels, this time can be only $\Theta(\log(n))$.

	LVPC [JLT20]	Elmo [KL]	Donner
Scripting requirements	Bitcoin	Bitcoin + ANYPREVOUT	Bitcoin
Multi-hop	✓	✓	✓
Secure against Domino attack	✗	✗	✓
Path privacy	✗	✗	yes
Time-based fee model	✓	✗	✓
Unlimited lifetime	✗	✓	✓
Storage Overhead per party	$\Theta(n)^*$	$\Theta(n^3)$	$\Theta(1)$
Off-chain closing	✓	✗	✓
Offload: txs on-chain	$\Theta(n)$	$\Theta(n)$	1
Offload: time delay	$\Theta(n)^*$	$\Theta(n)^*$	1

Conceptual advancements in this work. We show that rooted VCs are by design prone to severe drawbacks including the *Domino attack* (see Section 7.3), a new DoS/griefing style attack in which (i) a malicious intermediary of a VC or (ii) an attacker establishing a VC with itself over a number of honest PCs can close the whole path of underlying PCs and bring them on-chain. Not only are all existing Bitcoin-compatible

VC constructions affected by this attack, in fact the ideal functionalities against which they are proven secure do permit this attack, but also this attack is so severe that it can potentially shut down the underlying PCN, as we show in Section 7.3.3. As a result, we argue that none of the existing Bitcoin-compatible VC constructions should be deployed in practice. Furthermore, the rooted design allows adversaries to learn the identity of participants other than their direct neighbors, thereby breaking what we call *path privacy* (see Section 7.3.4). Given these security and privacy shortcomings, we introduce a paradigm shift towards the design of *non-rooted* VCs, based on two fundamental ingredients.

First, instead of being rooted, the VC is funded independently from the underlying PCs, by one of the VC endpoints. The underlying PCs are used to lock up some funds (or collateral) that are paid to the honest VC endpoint if the other VC endpoint misbehaves. We illustrate this concept on a high level in Figure 7.1(b.1). In contrast to rooted designs, VCs can be offloaded without having to close the underlying PCs, which is the key to prevent Domino attacks. Since the VC is only funded by one endpoint, only this funding endpoint has the means of transforming the VC to a PC (offload). Subsequently, the other one cannot get their money via offloading in case of misbehavior. This issue is solved by compensating the non-funding endpoint in case the funding endpoint has not transformed the VC to a PC within a channel lifetime T, see Figure 7.1(b.2) and (b.3).

This lifetime T is the second crucial aspect where we depart from the state of the art. Current solutions provide unlimited lifetime without guaranteeing however that the VC will remain open, as an intermediary node could initiate the offloading. Instead, our design ensures that the VC is open until time T, which can be repeatedly prolonged if all involved parties agree. This allows intermediaries to charge fees based on the lifetime of the VC, which corresponds to the time they have to lock up their funds, something that is not possible in current VC solutions with unlimited lifetime [KL]. The improvements over existing Bitcoin-compatible multi-hop VC constructions are summarized in Table 7.1. We compare with single-hop constructions and with those relying on Turing-complete smart contracts in Table F.2 in Appendix F.2.

Our contributions can be summarized as follows:

- We introduce the Domino attack, which allows the adversary to close arbitrarily many PCs of honest users, thereby destructing the underlying PCN. We argue that any rooted construction, in particular, all existing Bitcoin-compatible VC constructions are prone to this attack. We show the severity of this attack in a quantitative analysis; given current BTC transaction fees, it suffices for an attacker to spend 1 BTC to close every channel in the current LN. Even though VC protocols are not yet used in practice, we find it crucial to show this attack before any construction gets implemented, offering instead a secure alternative.
- We present Donner, a new VC protocol that departs from the rooted paradigm by funding the VC from outside of the underlying PC path. In addition to being

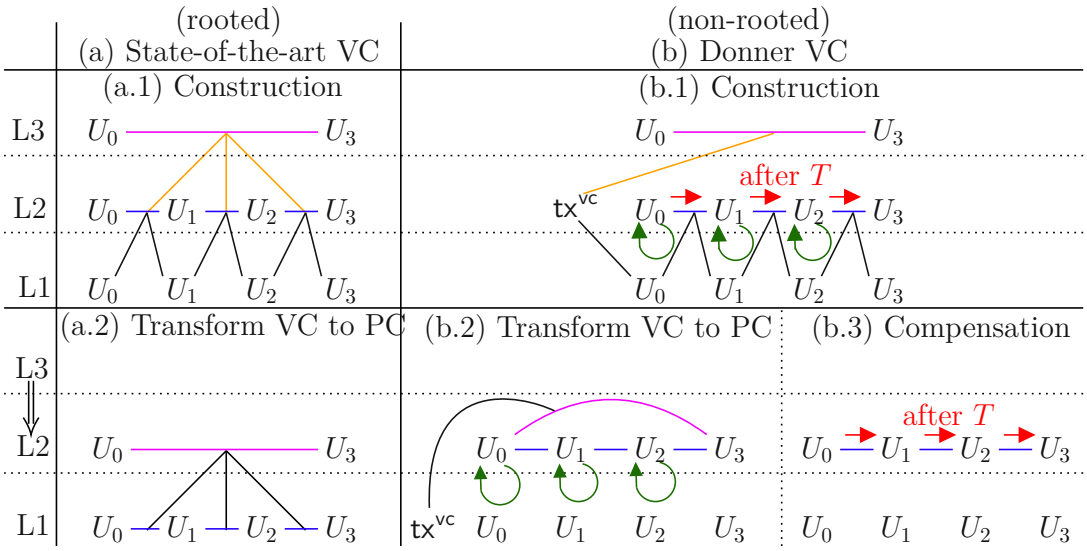

Figure 7.1: Conceptual comparison of (a) state-of-the-art VCs (rooted) and (b) our protocol (non-rooted) on layers L1 (blockchain), L2 (PCs) and L3 (VCs). Note that the VC in (a.1) is funded by all the underlying channels In (b.1), the VC is funded only by U_0, indirectly via a transaction $\mathsf{tx}^{\mathsf{vc}}$. Additionally, in (b.1), a payment is set up from U_0 to U_3, whose outcome depends on whether the VC is offloaded. Offloading, i.e., the act of forcefully transforming a VC (L3) to a PC (L2) in (a.2), requires that all the underlying PCs (L2) are put on-chain (L1). In (b.2), offloading the VC keeps the PCs open, posting only $\mathsf{tx}^{\mathsf{vc}}$ on-chain (L1). Since offloading enables U_3 to receive their funds, the payment is refunded then. However, since in (b), only U_0 can offload, U_3 is compensated (b.3) after a timeout T via a payment that is executed iff U_0 has not offloaded the VC (i.e., (b.2) did not happen).

secure against the Domino attack, it significantly improves in terms of efficiency and interoperability over state-of-the-art VC protocols (see Table 7.1).

- We introduce the notion of *synchronized modification*, a novel subroutine allowing parties to atomically change the value or timeout of a synchronization protocol, a contribution of independent interest. Synchronized modification, non-rooted funding, and the *pay-or-revoke* paradigm [AMSKM21] are the core building blocks of Donner.

- We conduct a formal security and privacy analysis of Donner in the Universal Composability framework.

- We conduct experimental evaluations to quantify the severity of the Domino attack and demonstrate that Donner requires significantly fewer transactions than state-of-the-art VCs; Donner decreases the on-chain costs for offloading VCs from linear in the path length to a single one and the storage overhead per PC from linear or

logarithmic in LVPC [JLT20] (depending on how the VC is constructed) or cubic in Elmo [KL] to constant.

7.2 Background and notation

7.2.1 UTXO based blockchains

We adopt the notation for UTXO-based blockchains from [AEE+21], which we shortly review next. In UTXO-based blockchains, the units of currency, i.e., the *coins*, exist in *outputs* of transactions. We define such an output as a tuple $\theta := (\mathsf{cash}, \phi)$; $\theta.\mathsf{cash}$ contains the amount of coins stored in this output and $\theta.\phi$ defines the condition under which the coins can be spent. The latter is done by encoding such a condition in the scripting language of the underlying blockchain. This can range from simple ownership, specifying which public key can spend the output, to more complex conditions (e.g., timelocks, multi-signatures, or logical boolean functions).

Coins can be spent with *transactions*, resulting in the change of ownership of the coins. A transaction maps a list of outputs to a list of new outputs. For better readability, we denote the former outputs as transaction *inputs*. Formally, we define a transaction body as a tuple $\mathsf{tx} := (\mathsf{id}, \mathsf{input}, \mathsf{output})$. The identifier $\mathsf{tx.id} \in \{0,1\}^*$ is assigned as the hash of the other attributes, $\mathsf{tx.id} := \mathcal{H}(\mathsf{tx.input}, \mathsf{tx.output})$. We model $\mathcal{H}$ as a random oracle. The attribute $\mathsf{tx.input}$ is a non-empty list of the identifiers of the transaction's inputs and $\mathsf{tx.output} := (\theta_1, ..., \theta_n)$ a non-empty list of new outputs. To prove that the spending conditions of the inputs are known, we introduce full transactions, which contain in addition to the transaction body also a witness list. We define a full transaction $\overline{\mathsf{tx}} := (\mathsf{id}, \mathsf{input}, \mathsf{output}, \mathsf{witness})$ or for convenience also $\overline{\mathsf{tx}} := (\mathsf{tx}, \mathsf{witness})$. Valid transactions can be recorded on the public ledger $\mathcal{L}$ called blockchain, with a delay of Δ. A transaction is valid if and only if (i) all its inputs exist and are not spent by other transactions on $\mathcal{L}$; (ii) it provides a valid witness for the spending condition ϕ of every input; and (iii) the sum of coins in the outputs is equal (or smaller) than the sum of coins in the inputs.

There are several conditions under which coins can be spent. Usually, they consist of a signature that verifies w.r.t. one or more public keys, which we denote as $\mathsf{OneSig}(\mathsf{pk})$ or $\mathsf{MultiSig}(\mathsf{pk}_1, \mathsf{pk}_2, ...)$. Additional conditions could be any script supported by the scripting language of the underlying blockchain, but in this paper, we only use relative and absolute time-locks. For the former, we write $\mathsf{RelTime}(t)$ or simply $+t$, which signifies that the output can be spent only if at least t rounds have passed since the transaction holding this output was accepted on $\mathcal{L}$. Similarly, we write $\mathsf{AbsTime}(t)$ or simply $\geq t$ for absolute time-locks, which means that the transaction can be spent only if the blockchain is at least t blocks long. A condition can be a disjunction of subconditions $\phi = \phi_1 \vee ... \vee \phi_n$. A conjunction of subconditions is simply written as $\phi = \phi_1 \wedge ... \wedge \phi_n$.

To visualize how transactions are used in a protocol, we use transaction charts. The charts are to be read from left to right. Rounded rectangles represent transactions, with

incoming arrows being their inputs. The boxes within the transactions are the outputs and the value in them represents the amount of output coins. Outgoing arrows show how outputs can be spent. Transactions that are on-chain have a double border (see, e.g., Figure F.4 in Appendix F.4.1).

7.2.2 Payment channels

Two users can utilize a payment channel (PC) in order to perform arbitrarily many payments while putting only two transactions on the ledger. On a high level, there are three operations in a PC operation: *open*, *update* and *close*. First, to open a channel, both users have to lock up some money in a shared output (i.e., an output that is spendable if both users give their signature) in a transaction called the *funding transaction* or tx^{f}. From this output, they can create new transactions called *state* or tx^{s} which assign each of them a balance. Once the funding transaction is on the ledger, the users can exchange arbitrarily many new states (balance updates) in an off-chain manner, thereby realizing the update phase of the channel. Once they are done, they can close the channel by posting the final state to the ledger.

In this work, we use PCs in a black-box manner and refer the reader to [AEE+21, MMSK+17,MMS+19] for more details. We abstract away from the implementation details and instead model the state of the channel as the outputs contained in a transaction tx^{s}, which is kept off-chain. For simplicity, we assume that this is the only state that the users can publish and abstract away from how the dishonest behavior is handled. In practice, it is possible that a dishonest user publishes a stale state of the channel and current constructions come with a way to handle this case (e.g., through a punishment mechanism that compensates the honest user [AEE+21]). We illustrate this abstraction in Figure 7.2.

7.2.3 Payment channel networks

A payment-channel network (PCN) [MMSK+17] is a graph where the nodes represent the users and the edges represent the PCs. The Lightning Network [PD16] is the state of the art in both PCs and PCNs for Bitcoin, and the largest PCN in terms of coins locked within its channel fundings, currently having around 81k channels, 19k active nodes, and a total capacity of 3k BTC (around 130M USD).

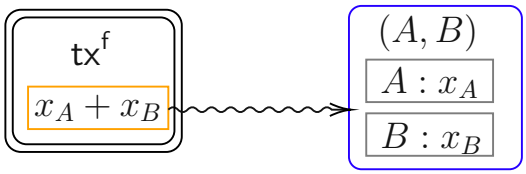

Figure 7.2: We abstract PCs using a squiggly line to hide details that are not needed in this work. $P : x_P$ indicates that user P owns x_P coins in the state tx^{s}, written as (A, B). The box containing $x_A + x_B$ indicates the shared output of A and B.

In a PCN, any two users connected by a path of channels can perform what is called a *multi-hop payment* (MHP). Assume that there is a sender U_0 who wants to pay α coins to a receiver U_n, but they do not have a direct channel. Instead, they are connected by a path of channels going through intermediaries $\{U_i\}_{i\in[1,n-1]}$, such that any pair of neighbors U_j and U_{j+1} have a channel $\overline{\gamma_j}$, for $j \in [0, n-1]$. A mechanism synchronizing all channels on the path is required for a payment, such that each channel is updated to represent the fact that α coins moved from left to right. We give an example in Figure F.5 in Appendix F.4.2.

7.2.4 Blitz

There exist many different MHP protocols that synchronize the updates of channels. In particular, the *Blitz* [AMSKM21] protocol is useful for this work. In Blitz, the PC updates are dependent on a transaction called $\mathsf{tx}^{\mathsf{er}}$, which acts as a global event. The PCs are synchronized in the following way: If $\mathsf{tx}^{\mathsf{er}}$ is posted on-chain, the updates are reverted, otherwise, they are successful. In other words, the sender sets up a MHP conditioned on a "refund enabling" transaction $\mathsf{tx}^{\mathsf{er}}$ in a way that the refund can be triggered, if anything goes wrong. If all channels participated honestly, the sender does not post $\mathsf{tx}^{\mathsf{er}}$ and the MHP goes through (see Figure 7.3). In a bit more detail, Blitz consists of four operations:

1. *Setup.* The sender U_0 creates a synchronization transaction $\mathsf{tx}^{\mathsf{er}}$ as depicted in Figure 7.3b, which has an output θ_{ϵ_i} holding ϵ coins for each user except the receiver U_n. The value ϵ is set to the smallest possible value that the underlying blockchain allows (ideally zero); these outputs are merely to enable other transactions.

2. *Open.* Each channel sequentially, from sender to receiver, sets up a payment whose success or refund is conditioned on a time T or transaction $\mathsf{tx}^{\mathsf{er}}$, as conceptualized in Figure 7.3a and shown in detail in Figure 7.3c. In a nutshell, two users U_i, U_{i+1} update their channel γ_i to a state where the amount to be paid α (more precisely α_i which encodes a per-hop fee) coming from U_i can be spent as follows: Either by U_{i+1} using $\mathsf{tx}^{\mathsf{p}}_i$ after time T or by U_i using $\mathsf{tx}^{\mathsf{r}}_i$ if $\mathsf{tx}^{\mathsf{er}}$ is posted on-chain. Since each $\mathsf{tx}^{\mathsf{r}}_i$ uses the corresponding output θ_{ϵ_i} of $\mathsf{tx}^{\mathsf{er}}$, the UTXO model ensures that $\mathsf{tx}^{\mathsf{r}}_i$ can only be posted if $\mathsf{tx}^{\mathsf{er}}$ has been posted before.

3. *Finalize.* After the receiver has successfully set up the payment, she sends back a confirmation to the sender containing $\mathsf{tx}^{\mathsf{er}}$. If the sender receives a confirmation containing the $\mathsf{tx}^{\mathsf{er}}$ she created in the *setup* phase within some time, she goes idle. Otherwise, she posts $\mathsf{tx}^{\mathsf{er}}$, initiating the refunds (see *respond*).

4. *Respond.* Every user U_i monitors the blockchain if $\mathsf{tx}^{\mathsf{er}}$ appears. In case it appears before T, the user will publish the refund transaction $\mathsf{tx}^{\mathsf{r}}_i$ for her channel γ_i. If the two users in γ_i collaborate, both updates and refunds can always be performed off-chain.

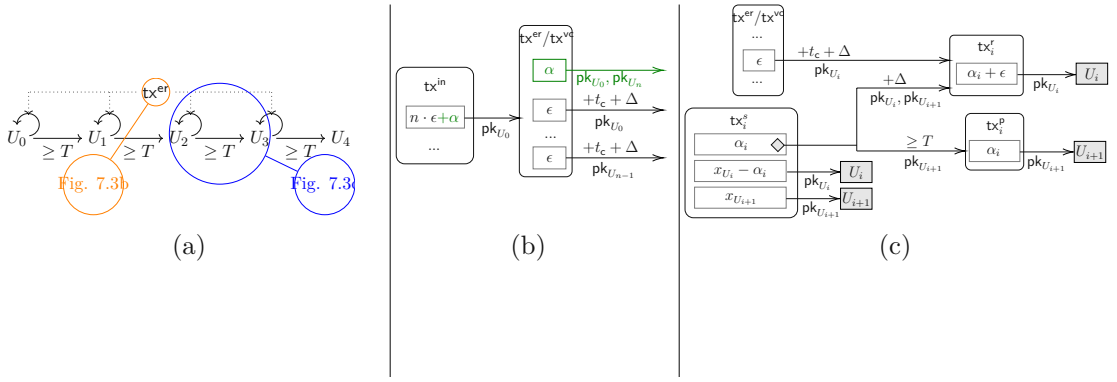

Figure 7.3: (7.3a) Illustration of the Blitz synchronization protocol; (7.3b) Off-chain synchronization transaction spending from an output under U_0's control and linking to the collateral in each channel. (i) Without the green part: $\mathsf{tx}^{\mathsf{er}}$ in Blitz. (ii) With the green part: $\mathsf{tx}^{\mathsf{vc}}$ used for funding the VC in this work; (7.3c) Two-party contract used within each channel

In this work, we utilize a slightly modified version of Blitz as a building block. We mark the modification in green in Figure 7.3b and describe it in Section 7.5.3.

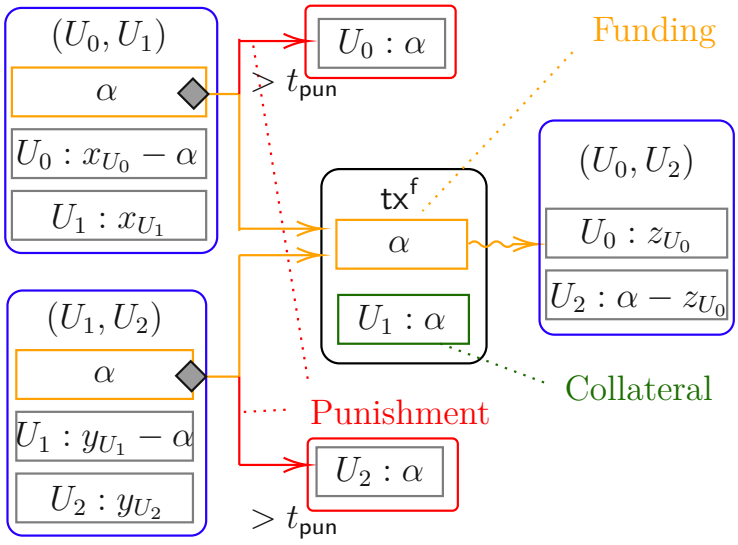

Figure 7.4: Illustration of a VC construction over a single intermediary. The VC funding tx^{f} is rooted in the underlying channels is the only way for the intermediary to get its collateral back. tx^{f} and the the punishment are mutually exclusive.

7.2.5 State-of-the-art virtual channels

A virtual channel (VC) allows two users to establish a direct channel, without putting any transaction on-chain. Indeed, the fundamental difference between a PC and a VC is that in a VC, the funding transaction tx^{f} does not go on-chain in the honest case. To still ensure that users do not lose their funds in case of dispute, this requires a new operation: In addition to the three operations *open*, *update* and *close* of PCs, we need the operation *offload*, which allows a user of the VC to put the funding transaction tx^{f} on-chain, transforming the VC into a PC in case of a dispute.

To understand how VCs work, let us look at an example following a state-of-the-art VC construction [JLT20]. This example is depicted in Figure 7.4. Assume U_0 and U_2 want to construct a VC via U_1, i.e., there exist PCs (U_0, U_1) and (U_1, U_2), and they wish to build a VC (U_0, U_2). To *open* a VC, the main idea is to take the desired VC capacity α and lock it in both channels, such that α coins come from U_0 and α coins from the intermediary U_1. These $2 \cdot \alpha$ coins are used both for funding the VC and as collateral; these coins can be spent in the following, mutually exclusive ways:

(i) by putting the funding transaction tx^{f} on-chain, which simultaneously funds the VC and refunds the intermediary its collateral α, or

(ii) if both α coins are not spent by a chosen punishment time t_{pun}, U_0 and U_2 can each claim α coins, which is the maximal amount they could hold in the VC

Clearly, U_1, who is part of both channels, is incentivized to put tx^{f} on-chain, as this is the only way to get her collateral back. Simultaneously, the two end-users U_0 and U_2, who are only part of one of the channels, are ensured that either tx^{f} goes on-chain, or else they receive the full α.

Putting tx^{f} on-chain is called *offloading* and is a safety mechanism to ensure that users can claim their rightful balance in case of a dispute. Offloading can be initiated by either U_0 or U_2 (by closing their respective channel and threatening to take the collateral if U_1 does not react), or by U_1 by simply closing both channels. We emphasize that the money of tx^{f} comes from both underlying channels, i.e., it can only exist on-chain if both underlying channels have been put on-chain (closed). We call this design a *rooted* VC.

If there is no dispute, the transactions depicted in Figure 7.4 remain off-chain and the underlying channels (U_0, U_1) and (U_1, U_2) remain open. The *update* of the VC requires no interaction of the intermediaries, the end-users simply update the channel (U_0, U_2) as they would a PC. Finally, to *close* the VC, the final balance of the VC has to be mapped into the base channels so that in the end both VC endpoints receive the latest balance of the VC and the intermediaries do not lose coins. Note that with the exception of *offload*, which requires *at least* one on-chain transaction (i.e., the funding), all other operations require no on-chain transaction. This single-intermediary idea can be used to construct a tree-like structure over a path of arbitrary intermediaries to get VCs of arbitrary length. We show this concept in Figure 7.5.

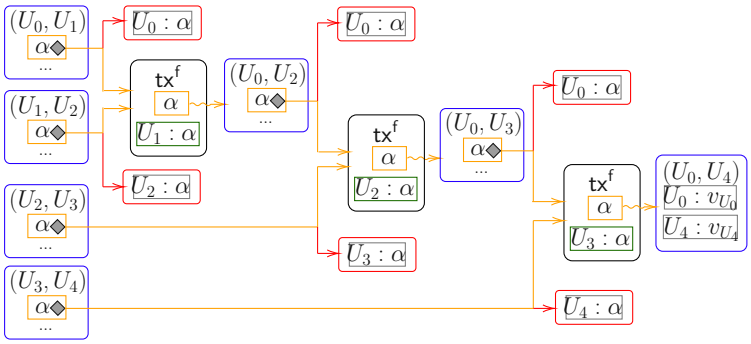

Figure 7.5: Illustration of a rooted VC via multiple hops. The yellow lines indicate how the VC is rooted. All transactions connected to and to the left of tx^{f} need to be put on-chain in case the rightmost VC is offloaded.

7.3 The Domino attack

7.3.1 Reasons that lead to the attack

Observation 1: Balance security for VC endpoints. Independently of its inner workings, any VC construction must ensure that honest VC endpoints Alice and Bob can cash out the coins they hold in the VC (i.e., get their coins on-chain). As discussed in Section 7.2.5, VCs are akin to payment channels (PCs), with the difference of having their funding transaction off-chain. This means that both endpoints can no longer directly claim their latest balance as in a PC. Instead, the VC funding transaction first needs to be put on-chain through the operation offload, which can be initiated by the VC endpoints and in some existing VC protocols [KL] even by the intermediaries.

Observation 2: VC funding transaction is rooted in all underlying base channels. We recall that to enable the offload operation, the VC funding takes as inputs (either directly or indirectly, via intermediate transactions) outputs of each of the underlying base channels. We denote such a VC as being *rooted* in the base channels.[2] At a first glance, this seems the most natural approach since it allows both endpoints to offload the VC and the intermediaries to unlock their collateral. However, a rooted funding implies that it can be posted on-chain *if and only if all underlying PCs are closed.* This feature is the source of the Domino attack, as shown next.

7.3.2 Attack description

The *Domino attack* is essentially a DoS or griefing style attack. It follows directly from the two observations mentioned above and can proceed in the following phases: (i) an

[2]By base channel we mean either a PC or a VC that was used for opening a VC, to capture the fact that VCs can be constructed recursively.

adversary controlling two nodes opens two PCs encasing a path of victim channels; (ii) the adversary opens a VC to herself via these victim paths; and (iii) she initiates the offloading of the VC.

The effect of this attack is to force the closure of every channel on this path, i.e., the two the attacker created and the channels on the victim path. Anyone not closing their channel risks losing their money. In stark contrast to payment protocols in PCNs such as Lightning or Blitz where closing one channel in the payment path still allows channels in the rest of the path to remain open, in current VC constructions there is no way that honest nodes can settle their channels honestly off-chain and keep them open. They are forced to close every channel, as the VC funding can only exist on-chain if all base channels are closed.

Example. Assume an attacker controlling nodes U_0 and U_4 who wants to perform a Domino attack on the victim path U_1, U_2 and U_3, see Figure 7.5. If not already opened, the attacker opens the channels (U_0, U_1) and (U_3, U_4). Then, she constructs a VC between her own nodes U_0 and U_4 recursively, as, e.g., established in the LVPC protocol [JLT20]. After the attacker is done with this step, the transaction structure among different users is as in Figure 7.5. The attacker can now unilaterally force the closure of all underlying channels, i.e., the PCs (U_0, U_1), (U_1, U_2), (U_2, U_3) and (U_3, U_4) as well as the intermediate VCs (U_0, U_2), (U_0, U_3) and the offloading of (U_0, U_4).

First, U_4 closes the PC (U_3, U_4), which she can do on her own. In the rooted VC example of Figure 7.5 (e.g., this could be LVPC), the output in the state of (U_3, U_4) which is used to fund the VC (U_0, U_4) goes to U_4, *unless* it is first consumed by the VC. This means that an honest U_3 will lose money in the channel (U_3, U_4) to U_4 by means of the punishment transaction on the bottom right in Figure 7.5 (dubbed *Punish* transaction in the LVPC protocol), unless she closes the channel (U_0, U_3) and claims its money by posting tx^{f}, i.e., the transaction funding the VC (i.e., offloading) (U_0, U_4), dubbed *Merge* transaction in LVPC.

However, to post tx^{f} for (U_0, U_4), U_3 first needs close (U_0, U_3). U_3 initiates the offloading by first closing (U_2, U_3). This triggers a similar response from U_2, who is now at risk of losing the coins in (U_2, U_3), unless she offloads (U_0, U_3) by putting the corresponding tx^{f}. But to do that, U_2 first needs to close (U_0, U_2). This is done, finally, by closing (U_1, U_2), which forces U_1 to close also (U_0, U_1).

In the end, all channels are closed (as shown in Figure F.1 in Appendix F.3). Let us clarify that by closing the underlying channels we mean that at least two transactions per channel have to be put on-chain, one for closing the channel and another one to spend the collateral locked for the VC. Due to the fact that LVPC first splits the channel into two subchannels before using one of them to fund the VC, closing the initial channel simultaneously spawns a new channel (i.e., the remaining subchannel) that has a capacity reduced by the amount put in the collateral funding the VC. The Domino attack works regardless of how the recursion was applied, as well as on Elmo [KL]. In LVPC some (U_3 in the example above) and in Elmo all intermediaries can carry out this attack. The Domino

attack can also be launched if the attacker controls only one of the endpoints, assuming the other one agrees to open a VC with her over the victim path. We remark that LVPC and Elmo are modeled in the UC framework, however, their ideal functionalities explicitly allow for the Domino attack.

7.3.3 Quantitative analysis of the Domino attack

To quantify the severity of the Domino attack, we perform the following simulation. We take a current (March 2022) snapshot of the Lightning Network (LN) [LN 22]. In this snapshot, there are 83k channels, 20k nodes, and 3284 BTC (around 150M USD) locked in channels (of the largest connected component). The nodes' connectivity varies in the LN. There are leaf nodes having only one open channel, and there are nodes with almost 3000 channels. Additionally, entities can control multiple nodes. The entities can be linked by their alias, as pointed out in [RVMS+21], something we follow in this simulation as well.

Clearly, differently connected nodes can launch the Domino attack with more or less devastating effects. The better connected a node is, the more channels can be closed down. Note that for this attack, it does not matter how many coins are locked in the channels under the control of the attacker and not even the number of nodes the attacker controls, but instead the number of open channels and the kind of paths that exist to another node under the attacker's control; the source and destination may be the same node.

Analyzing existing nodes. To measure the damage that can be caused by existing nodes in the LN with two or more open channels, we do the following. Assuming each node, or more precisely, each alias, is performing the Domino attack. This means, using the open channels the attacker tries to close as many channels as possible. Computing the optimal set of VCs the attacker would need to open to maximize the channels is computationally expensive and out of the scope of this simulation. Instead, we settle for a simpler heuristic. The attacker computes the cycle basis for a root node controlled by the attacker, yielding paths starting and ending at one node under the attacker's control. The attacker chooses the longest one and proceeds to close the channels by performing the Domino attack. Now, on the new network with fewer channels, the attacker repeats these steps, until all of the attacker's channels are closed and they can do no further damage.

We count the channels an attacker can close with this approach for each alias. Each node can close 1284 channels on average, which amounts to around 1.5% of all channels in the LN. However, note that around 8% of all nodes can close no channels at all, while the most well-connected entity can close around 53k channels, which is more than 60% of the LN. We visualize our results in Figure 7.6a, where for a given interval of how many channels an entity can close, we show the percentage of nodes that fall into this category. The source code and raw results of this simulation can be found at [Sim22].

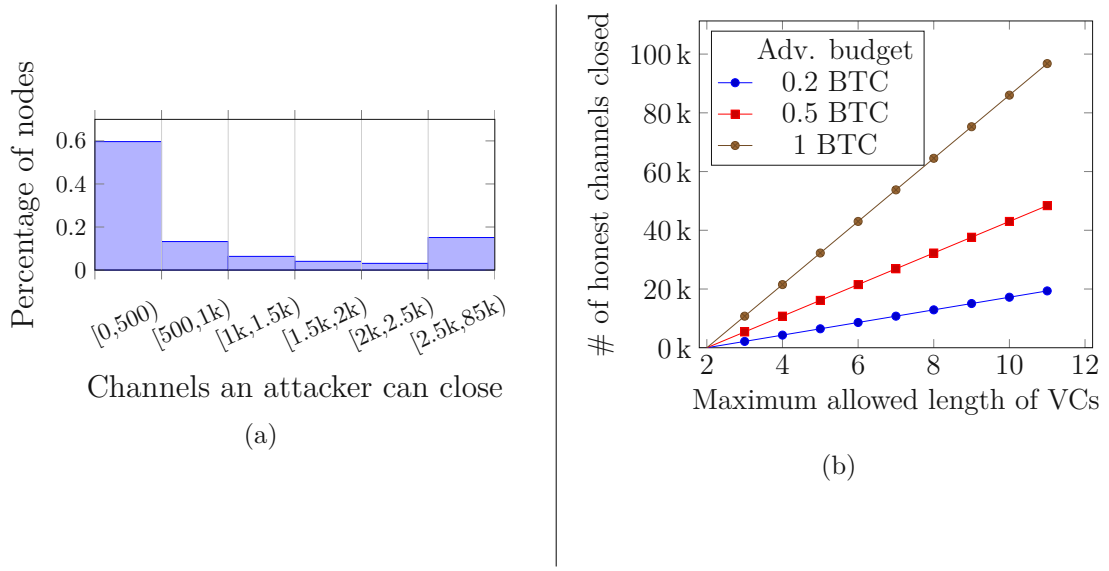

Figure 7.6: Simulated effect of the Domino attack.

To make matters worse, an attacker can target specific channels with this. This allows the attacker to perform attacks similar to *Route Hijacking* [TZS20], a DoS attack where an attacker strategically places a channel in a topologically important location and announces low fees. Subsequently, users will route their payments through the attacker's channel who can then drop the requests. In the worst case, this can (temporarily) disconnect parts of the network from one another. In the Domino attack, an attacker can disconnect parts of the network directly, by closing all edges that connect the two subgraphs.

Analyzing newly placed nodes. In this second analysis, we let the attacker create new channels instead of assuming an existing node is corrupted. Clearly, without any restrictions, an attacker can do more damage than in the previous simulation, i.e., by opening the same (and more) channels as the best-performing node which had a bit less than 3000 channels. Taking a current average fee of 0.000031 BTC (1.27 USD) per transaction [Bit22a], this would cost an adversary around 0.186 BTC. In more detail, 0.093 BTC are needed for opening these channels and again 0.093 BTC for closing them after establishing the according VC, triggering the Domino attack. Note that the latter amount is also needed if the channels are already there (in the previous simulation).

We therefore put some restrictions on the attacker. We assume that an adversary has a certain budget to spend on fees for establishing channels over the network. Further, the adversary constructs VCs of a length of up to $n \in [2, 11]$ to herself, i.e., the adversary is the first and last node. We set the maximum VC length n to 11, the diameter of the LN snapshot, i.e., at this length, every node can reach every other node.

The adversary needs to post 3 on-chain transactions per VC with the associated fees, two for establishing the two PCs encasing the victim path and one to close one of these

channels. Further, for the VC itself, a certain minimum amount is needed to open it, similar to LN payments. However, since this amount is presumably not only very small but also the adversary gets it back, we omit it in our simulation and say instead that the adversary performs this attack in sequence. Finally, we note that the effect of this attack is likely to be even more severe in reality, since in existing VC constructions, not only does the channel need to be closed, but subsequent transactions making up the rooted funding of the VC need to be posted as well.

We present our results in Figure 7.6b. Using only 1 BTC for fees, the adversary can close up to 97k honest channels, which is more than all channels in our LN snapshot (83k), and cause a cost of at least 6 BTC to the involved nodes. Budgets in the order of 0.2, 0.5, and 1 BTC are not unrealistic, as there are 1501, 799, and 453 nodes, respectively, holding this money within the LN, assuming equal balance distribution in the channels, i.e., 0.5% of nodes in the LN have enough balance to shut down the whole network. If we consider all Bitcoin addresses (even outside the LN), there exist 815k addresses owning 1 BTC or more [Bit22c].

We remark that since VCs are not used in practice, we cannot evaluate this in the real world. However, previous work has already shown the feasibility of similar DoS or griefing attacks and how they transfer to the real world [HZ20]. For a discussion on why it is infeasible to deter this attack with fees, we refer to Appendix F.2. From our simulation, it follows that this attack is too severe for the adaption of current VC solutions in PCNs such as the LN. In order to make VCs usable in practice, it is essential to prevent the Domino attack.

7.3.4 More drawbacks of current VC constructions

Unlimited lifetime. Existing VC constructions such as Elmo [KL] offer VCs with an a priori unlimited lifetime. On a high level, unlimited lifetime of a VC means that if every party agrees (including endpoints and intermediaries), the VC can remain open potentially forever. While existing work highlights unlimited lifetime as a desirable feature for both PCs and VCs, we view it as a drawback in the context of VCs. Indeed, there is an important difference between VCs and PCs: in a VC funds are locked up not only by the endpoints but also by the intermediaries of the underlying path. Without a lifetime, intermediaries could have their collateral locked up forever unless they decide to go on-chain, which however forces them to close their PCs. Related to that, intermediaries should charge a fee proportional to the collateral and the time this collateral is locked (analogously to the LN): without a lifetime, the second parameter cannot be estimated nor enforced without closing the base PCs.

We, therefore, propose a new approach: instead of having an a priori unlimited lifetime, we fix a certain lifetime at the point of creation. When this lifetime expires, users have the option to prolong it for another fixed lifetime if everyone agrees or to close it. Prolonging it means that the VC remains active and any applications hosted on top of can be kept on being used smoothly. In addition, every intermediary can charge a lifetime-based fee

every time they prolong the VC. While all agree, they can repeat this process indefinitely. If one party wants to stop it, the party can unlock their funds *without having to close any channel on-chain*. We explain this concept in more detail in Section 7.4.

Recursiveness. The last issue we point out comes from how the VC funding is rooted in the underlying channels. In current VC constructions, the VC funding is built by recursively combining two channels at a time, forming a tree with the VC funding transaction being the root of the tree and the underlying channels being the leaves. This has two negative implications. First, in addition to closing all PCs (which requires at least one on-chain transaction per channel), i.e., the leaves of the tree, a linear number of transactions needs to go on-chain in order to offload a channel, i.e., the non-leaf nodes of the tree. Second, depending on how the recursiveness has been applied, the time it takes to offload a VC is also either linear (in case of an unbalanced tree, cf. Figure F.6 in Appendix F.5.1) or logarithmic (in case of a balanced tree, cf. Figure F.7 in Appendix F.5.1) in the number of underlying channels. In our construction, offloading involves only a constant number of on-chain transactions as elaborated in the next section.

Lack of path privacy. State-of-the-art VC constructions create the rooted funding by connecting outputs of pairs of channels in a recursive way. However, this requires the interaction of some intermediaries with more than their direct neighbors on the path. In our construction, intermediaries on the path only learn about their direct neighbors in the honest case, exactly as in the Lightning Network.

7.4 Donner: Key ideas

We describe the core ideas of Donner by assuming that a slight variant of the previously described Blitz construction is used as the underlying MHP protocol. As detailed below, our construction is parameterized over it, so that other functionality-equivalent MHP protocols could be deployed instead.

High level architecture. Let us assume U_0 and U_n, connected via U_i for $i \in [1, n-1]$, wish to open a bidirectional VC with capacity α and time T fully funded by U_0. First, U_0 starts with a slightly modified version of the Setup phase of a Blitz payment of α coins, as explained in Section 7.2.4, let us call it Setup*. In this modified phase, U_0 proceeds to create a transaction $\mathsf{tx}^{\mathsf{vc}}$ as depicted in Figure 7.3b (this time, including the green part) instead of $\mathsf{tx}^{\mathsf{er}}$. $\mathsf{tx}^{\mathsf{vc}}$ takes an input from U_0 and creates an output holding α coins and like in the Setup phase, an output holding ϵ coins for each user except the receiver U_n. This transaction will serve two purposes: (i) it will be the funding of the VC and (ii) it will be used to synchronize a Blitz payment.

Next, U_0 and U_n proceed to create the initial state (see Section 7.2.2) tx^s of the VC using $\mathsf{tx}^{\mathsf{vc}}$ as funding. We emphasize that this process is exactly the same as for a PC, the only difference being that the funding transaction $\mathsf{tx}^{\mathsf{vc}}$ has these additional outputs holding ϵ and we do not intend to publish $\mathsf{tx}^{\mathsf{vc}}$ on-chain. After this step is successful, U_0 initiates the remaining phases of Blitz (Open, Finalize, and Respond) using $\mathsf{tx}^{\mathsf{vc}}$. After

completion, a Blitz payment of value α is open between U_0 and U_n conditioned on $\mathsf{tx}^{\mathsf{vc}}$, i.e., it is refunded if $\mathsf{tx}^{\mathsf{vc}}$ is posted and otherwise successful after time T.

Intuition security. At this point, the VC is considered open and can be used exactly like a PC. The careful reader might be wondering why this VC is safe to use. After all, we detached the funding from the underlying PCs and removed the receiver U_n's ability to offload the VC. However, the sender U_0 did set up a Blitz payment to U_n of α coins, which is the full capacity of the VC. By putting the VC funding inside the synchronization transaction of Blitz, we make the two actions *offload the VC* and *refund the Blitz payment* atomic. In other words, if U_0 does not offload, U_n will automatically receive the full VC capacity via the payment after T.

Getting rid of the Domino attack. We recall the causes for the Domino attack: (i) the VC funding has to be enforceable on-chain by offloading and (ii) the VC funding is rooted in all underlying PCs. To prevent the attack, we got rid of (ii): The funding $\mathsf{tx}^{\mathsf{vc}}$ comes solely from U_0, i.e., it is independent (or detached) from the PCs underlying the VC. The VC can be offloaded without closing the underlying PCs, simply by U_0 posting $\mathsf{tx}^{\mathsf{vc}}$. Once posted, all PCs can be honestly settled, updating the PC to reflect the refund or the success of the Blitz payment, as in Blitz itself or other synchronization protocols.

Closing the VC. One of the most essential operations of the VC operation is closing the VC honestly, i.e., off-chain. This is challenging because closing needs to proceed in a way, such that no one is at risk of losing funds. To solve this challenge, we first observe that if the receiver U_n already owns all α coins in the VC, the VC endpoints need merely wait until the Blitz timeout T runs out. At this point, the Blitz payment will be successful automatically. But what about when U_n owns $0 \leq \alpha' < \alpha$ coins in the VC? We need a protocol that atomically changes the value of the Blitz transaction from α to α'. To solve this issue, we introduce a new protocol, called *synchronized modification*, which given a payment of value α tied to transaction $\mathsf{tx}^{\mathsf{vc}}$ and a timeout T, allows for updating the payment to a value α' such that $0 \leq \alpha' < \alpha$. This is illustrated in Figure 7.7.

Synchronized modification works as follows. We can update individual 2-party Blitz contracts to the new value α' from right to left. An intermediary U_i is sure to not lose money, because the atomicity of Blitz ensures that in both the left (U_{i-1}, U_i), having locked α, and the right channel (U_i, U_{i+1}), having locked α', the payment is either refunded or succeeds. In the former case, U_i does not lose money, as both payments are reverted. In the latter case, U_i gains α while paying α', so U_i gets some money. We can incentivize the participation of intermediary users with fees. Alice is incentivized to publish $\mathsf{tx}^{\mathsf{vc}}$ if the correct updates do not reach her (paying more money than she owes otherwise), thereby ensuring the atomicity of the synchronized modification. If all the channels are updated, they can simply go idle waiting for the payment to be successful after T, or they can finalize this payment instantly by using the fast track functionality [AMSKM21].

Fair unlimited lifetime. The timeout parameter T serves an additional purpose here: It is the lifetime of the VC. VC endpoints need to close the VC before T expires.

Interestingly, we can use the aforementioned *synchronized modification* operation also for extending this lifetime. In particular, besides updating the contracts in each channel to a smaller amount, as shown in Figure 7.7, we can in fact update the timeout T in each channel. Before the initial timeout T expires, the VC endpoints can run a synchronized modification update from receiver to sender. If everyone agrees, they can update to the time $T' > T$, and intermediaries would charge a fee for this. Intuitively, users are incentivized to agree as they are fine to pay their money later (at T') to their right while receiving it earlier (at T) on their left. This solves the problem of the a priori unlimited lifetime of prior VC constructions. The endpoints have the guarantee that the VC remains virtual until a pre-defined timeout, while the intermediaries have a guarantee that they can unlock their collateral after at most a pre-defined timeout without going on-chain and they can prolong it if everyone agrees for as long as they wish. Since the time for which the VC is prolonged is known, intermediaries can adopt a fee model that is based on time, which is not possible in existing solutions.

7.5 Donner: Protocol description

7.5.1 Security and privacy goals

We informally define three security and three privacy goals for our VC construction. For a formal definition of these properties as cryptographic games (Definitions 21 to 26) and

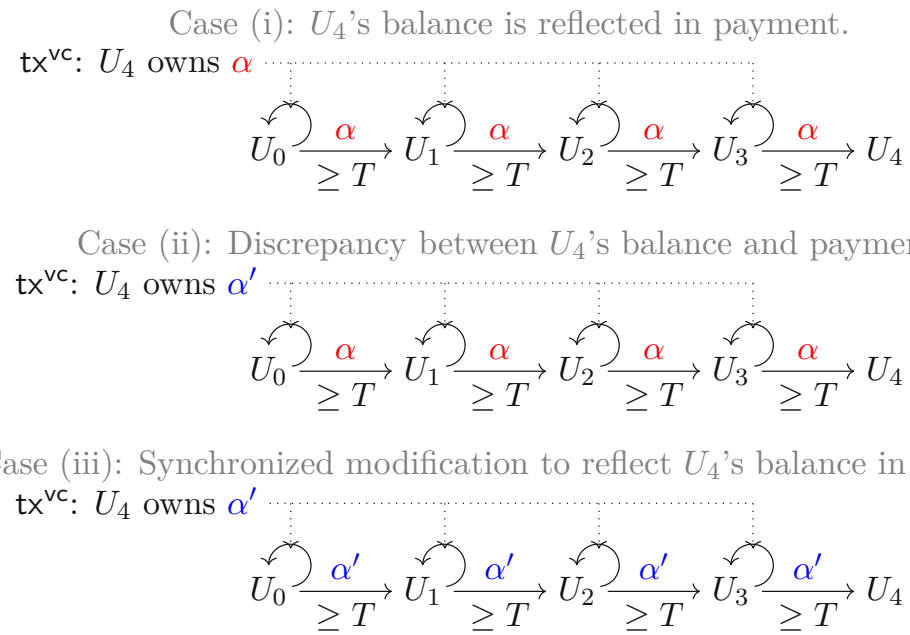

Figure 7.7: Synchronized modification: Safely modify the contract tied to a transaction $\mathsf{tx}^{\mathsf{vc}}$ in each channel atomically. Note that $\mathsf{tx}^{\mathsf{vc}}$ is the same transaction in all three cases.

proofs (Theorems 19 to 24), we defer the reader to Appendix F.6.6. We mark security goals with an S and privacy goals with a P. Side channel attacks (e.g., *probing* and *balance discovery*) constitute a significant privacy threat for PCNs [KYP+21]. Here, we rule out side channels from the attacker model to reason about the leakage induced by the design of the VC construction itself.

(S1) Balance security. Honest intermediaries do not lose any coins when participating in the VC construction.

(S2) Endpoint security. No user can steal the sender's balance of the VC. Additionally, the receiver is always guaranteed to get at least its VC balance.

(S3) Reliability. No (possibly colluding) intermediaries can force two honest endpoints of a VC to close or offload the VC before the lifespan T of the VC expires.

(P1) Endpoint anonymity. In an optimistic VC execution, intermediaries cannot distinguish if their left (right) user is the sending (receiving) endpoint or merely an honest intermediary connected to the sending (receiving) endpoint via other, non-compromised users.

(P2) Path privacy. In an optimistic VC execution, intermediaries do not learn any identifiable information about the other intermediaries, except for their direct neighbors.

(P3) Value privacy. The users on the path learn only about the initial and the final balance of the VC, not the value of the individual payments.

The careful readers may have noticed that P1 and P2 hold only for the optimistic case. Indeed, like in any other off-chain protocol (e.g., the Lightning Network), the channels have to go on-chain in order to resolve disputes in the worst case. This means that anyone observing the blockchain can reconstruct the path. Note, however, that this happens rarely, as the optimistic case is less costly for the participants. Designing off-chain protocols that achieve privacy even in case of disputes is an interesting open question.

7.5.2 Assumptions and prerequisites

Digital signatures. A digital signature scheme is a tuple of three algorithms $\Sigma := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy})$. On a high level, $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$ is a PPT algorithm that on input a security parameter λ generates a keypair $(\mathsf{pk}, \mathsf{sk})$. The public key pk is publicly known, while the secret key sk is only known to the user who generated that keypair. $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ is a PPT algorithm that on input a secret key sk and a message $m \in \{0,1\}^*$ generates a signature σ of m. Finally, $\{0,1\} \leftarrow \mathsf{Vrfy}(\mathsf{pk}, \sigma, m)$ is a DPT algorithm that on input a public key pk, a message m and a signature σ outputs 1 iff the signature is a valid authentication tag for m w.r.t. pk. We use a EUF-CMA secure [GMR88] signature scheme Σ as a black box throughout this work.

Payment channel notation. We model each payment channels as a tuple: $\overline{\gamma} := (\mathsf{id}, \mathsf{users}, \mathsf{cash}, \mathsf{st})$. The attribute $\overline{\gamma}.\mathsf{id} \in \{0,1\}^*$ uniquely identifies a channel; $\overline{\gamma}.\mathsf{users} \in \mathcal{P}^2$

identifies the two parties involved in the channel out of the set of all parties $\mathcal{P}$. Moreover, $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}_{\leq 0}$ denotes the total monetary capacity (i.e., the coins) of the channel and the current state is stored as a vector of outputs of $\mathsf{tx}^{\mathsf{state}}$: $\overline{\gamma}.\mathsf{st} := (\theta_1, ...\theta_n)$. In this work, we use channels in paths from a sender to a receiver. For simplicity, we say that $\overline{\gamma}.\mathsf{left} \in \overline{\gamma}.\mathsf{users}$ refers to the user closer to the sender, while $\overline{\gamma}.\mathsf{right} \in \overline{\gamma}.\mathsf{users}$ refers to the user closer to the receiver. The balance of both users can always be inferred from the current state $\overline{\gamma}.\mathsf{st}$. For convenience, we say that $\overline{\gamma}.\mathsf{balance}(U)$ gives the coins owned by $U \in \overline{\gamma}.\mathsf{users}$ in this channel's latest state $\overline{\gamma}.\mathsf{st}$. Finally, we define a channel skeleton γ for a channel $\overline{\gamma}$, as $\gamma := (\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users})$.

Ledger and channels. We use the ledger (or blockchain) and a PCN (both introduced in Section 7.2) as black-boxes in our construction. The ledger keeps a record of all transactions and balances and is append-only. The PCN supports opening, updating and closing of PCs. We assume the PCs involved in VCs to be already open. We interact with ledger and PCN through the following procedures.

$\mathsf{publishTx}(\overline{\mathsf{tx}})$**:** The transaction $\overline{\mathsf{tx}}$ is posted on-chain after at most Δ time (the blockchain delay), if it is valid.

$\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$**:** This procedure initiates an update in the channel $\overline{\gamma_i}$ to the state $\mathsf{tx}_i^{\mathsf{state}}$, when called by a user $\in \overline{\gamma_i}.\mathsf{users}$. The procedure terminates after at most t_{u} time and returns $(\mathsf{update{-}ok})$ in case of success and $(\mathsf{update{-}fail})$ in case of failure to both users. We call this function also to update our VC hosted on $\mathsf{tx}^{\mathsf{vc}}$.

$\mathsf{closeChannel}(\overline{\gamma_i})$**:** This procedure closes the channel $\overline{\gamma_i}$, when called by a user $\in \overline{\gamma_i}.\mathsf{users}$. The latest state transaction $\mathsf{tx}_i^{\mathsf{state}}$ appears on the ledger after at most t_{c} time.

$\mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, \mathsf{index}, U_0, U_n)$**:** Pre-creates the VC $\overline{\gamma_{\mathsf{vc}}}$, exchanging the initial state transactions with the other user in $\overline{\gamma_{\mathsf{vc}}}.\mathsf{users} := (U_0, U_n)$ based on the output identified by index of the funding transaction $\mathsf{tx}^{\mathsf{vc}}$ that remains off-chain for now. It finally returns $\overline{\gamma_{\mathsf{vc}}}$.

Assumptions and remarks. In our construction, we assume that every user U has a public key pk_U to receive transactions. Additionally, we assume that honest parties stay online for the duration of the protocol, like in the Lightning Network. A path-finding algorithm to identify a payment path can be called by $\mathsf{pathList} \leftarrow \mathsf{GenPath}(U_0, U_n)$. This will return a path in the PCN from U_0 to U_n. Path-finding algorithms are orthogonal to the problem tackled in this paper and we refer the reader to [SVR+20, RMKG18] for more details. Finally, we assume fee to be a publicly known value charged by every user. Note that in practice, every user can charge an individual fee. We reuse the pseudo-code definitions of Setup, Open, Finalize, and Respond from [AMSKM21] in Figure 7.8.

7.5.3 Detailed construction and pseudocode

Recall the setting, where U_0 and U_n, connected via U_i for $i \in [1, n-1]$, wish to open a bidirectional VC with capacity α fully funded by U_0. We consider the different phases

of Donner: OpenVC, UpdateVC, CloseVC, ProlongVC and Respond. We show the used macros in Figure 7.8(a), the procedure for updating individual PCs for the close or prolong VC phase in Figure 7.8(b), and the whole protocol in Figure 7.8(c). For completeness, we explain the protocol including the operations of Blitz [AMSKM21] below in prose, while in Figure 7.8(c) we show a modularized protocol based on the operations *setup*, *open*, *finalize* and *respond*. We remark that in this work, we could use any other construction providing the same functionality, e.g., this can be achieved by smart contract enabling UTXO-based chains such as the EUTXO model used in Cardano [CCM+20]. For better readability we simplify the protocol, e.g., we omit ids required for routing VCs concurrently. For the formal protocol description in the UC framework, we defer to Appendix F.6.4.

OpenVC. This phase uses a modified Blitz Setup phase (Setup*) and Open/Finalize of Blitz. *Setup*:* The sender U_0 starts by creating a transaction $\mathsf{tx}^{\mathsf{vc}}$ that contains an output θ_{vc} holding α coins spendable under the condition $\mathsf{MultiSig}(U_0, U_n)$ and n outputs θ_{ϵ_i} holding ϵ coins each spendable under the condition $\mathsf{OneSig}(U_i) + \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$, one for every user U_i for $i \in [0, n-1]$. Spending from θ_{vc}, U_0 and U_n create commitment transactions for the VC with $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$. This function pre-creates the VC $\overline{\gamma_{\mathsf{vc}}}$, exchanging the initial state transactions with the other user in $\overline{\gamma_{\mathsf{vc}}}.\mathsf{users} := (U_0, U_n)$ based on the output with index 0 of the funding transaction $\mathsf{tx}^{\mathsf{vc}}$ that remains off-chain for now. It finally returns $\overline{\gamma_{\mathsf{vc}}}$.

Open (Blitz): Then, each pair of users from U_0 to U_n performs `pcSetup` of [AMSKM21], which we briefly summarize as follows. Sender U_0 presents its neighbor U_1 with $\mathsf{tx}^{\mathsf{vc}}$ and an update of their channel to a state, where α coins of U_0 are spendable under the condition $\phi = (\mathsf{OneSig}(U_1) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(U_0, U_1) \wedge \mathsf{RelTime}(\Delta))$. Passing along $\mathsf{tx}^{\mathsf{vc}}$ does not violate privacy, due to the usage of stealth addresses, see Appendix F.5.2.

Before actually updating the channel, U_1 gives U_0 its signature for $\mathsf{tx}^{\mathsf{r}}_0$. $\mathsf{tx}^{\mathsf{r}}_0$ takes as inputs the output holding α of the aforementioned proposed state update and the output θ_{ϵ_0} of $\mathsf{tx}^{\mathsf{vc}}$ holding ϵ under U_0's control. After receiving the signature, they perform this update and revoke their previous state. In the same fashion, U_1 continues this procedure with its neighbor U_2 and this continues with its neighbor until the receiver U_n has successfully updated its channel with its left neighbor U_{n-1}. Then, U_n sends a confirmation to U_0 (*Finalize*).

UpdateVC. At this point the VC $\overline{\gamma_{\mathsf{vc}}}$ is considered to be open and ready to be used. An update is performed by creating a new state $\mathsf{tx}^{\mathsf{state}}_i$ and calling $\mathsf{updateChannel}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{state}}_i)$. This function updates the VC $\overline{\gamma_{\mathsf{vc}}}$, changing the latest state transaction to $\mathsf{tx}^{\mathsf{state}}_i$ and revoking the previous one. In case of a dispute, the users wait until the VC is offloaded. At this time, the VC is closed.

In the beginning, the whole balance lies with U_0, but once the balance is redistributed, the channel is usable in both directions. Should they wish to construct a channel where they both hold some balance initially, they can start the construction in the other direction for a second time, as we discuss in Appendix F.2. When they have rebalanced the money

inside the VC and definitely before time T, they proceed to the next phase, the closing phase.

CloseVC/ProlongVC (Synchronized modification). For closing the VC, assume its final balance is $\alpha - \alpha'$ belonging to U_0 and α' to U_n (and $T' = T$). For prolonging the lifetime, assume the new time is $T' > T$ (and $\alpha' := \alpha$). In either case, pairs of users perform the new functionality `2pModify` from right to left, which we outline as follows. U_n starts the following update process with its left neighbor U_{n-1}. U_n presents a state, where (instead of α) only α' coins from U_{i-1} are spendable under the condition $\phi = (\mathsf{OneSig}(U_n) \wedge \mathsf{AbsTime}(T)) \vee (\mathsf{MultiSig}(U_{n-1}, U_n) \wedge \mathsf{RelTime}(\Delta))$ (closing) or the time in this condition is changed to T' (prolong). For this new state, U_n creates a transaction $\mathsf{tx}^{\mathsf{r}}_{n-1}$ spending this output and the output of $\mathsf{tx}^{\mathsf{vc}}$ belonging to U_{n-1} and gives its signature for this new $\mathsf{tx}^{\mathsf{r}}_{n-1}$ to U_{n-1}. After U_{n-1} checks that the new state and new $\mathsf{tx}^{\mathsf{r}}_{n-1}$ are correct, they update their channel to this new state and revoke the previous one (cf. Figure 7.8(b)).

User U_{n-1} continues this process with its left neighbor U_{n-2} and so on until the sender U_0 is reached. U_0 checks that the balance in the state update is actually the balance that U_0 owes U_n in the VC, α'. If it is not the same, or no such request reaches the sender, U_0 simply publishes $\mathsf{tx}^{\mathsf{vc}}$ on-chain and claims $\mathsf{tx}^{\mathsf{r}}_0$ before the currently active timeout T expires. In the case where the correct request reaches the sender, they can either continue using the VC until T' (prolong) or in the case of closing, they wait until T expires, at which the money α' automatically moves from left to right to the receiver, or they perform the fast-track mechanism of [AMSKM21] to immediately unlock their funds (cf. Appendix F.2). VC endpoints do not need to wait until T, but can close the VC well before if they wish to do so.

Respond. This phase corresponds to the phase with the same name of Blitz, which proceeds thus. Participants have to monitor the ledger if $\mathsf{tx}^{\mathsf{vc}}$ is published. In case it is published and its outputs are spendable before T, each user U_i for $i \in [0, n-1]$ needs to refund the money they staked in their right channel. They can either do this off-chain if their right neighbor is cooperating or in the worst case, forcefully on-chain via $\mathsf{tx}^{\mathsf{r}}_i$. Similarly, after time T has expired without $\mathsf{tx}^{\mathsf{vc}}$ being published on-chain, each user U_i for $i \in [1, n]$ can claim the money from their left channel. Again, this can happen honestly off-chain or forcefully via $\mathsf{tx}^{\mathsf{p}}_i$.

Remarks. Because we detached the funding transaction from the underlying channels, we additionally get rid of the other issues presented in Section 7.3.4. Since the funding can be published independently from the channels and the collateral outcome depends on the funding, we give back the possibility to intermediaries to resolve their channels honestly. Additionally, as the funding is not constructed by combining the outputs of the underlying channels in sequence, we eliminate the additional linear on-chain transactions (needing only one) and reduce the linear (or logarithmic) time delay for publishing the funding transaction to a constant. Further, as we discuss in Section 7.6, Donner achieves

(a) Macros: $\texttt{genState}(\alpha_i, T, \overline{\gamma_i})$: Generates and returns a new channel state carrying transaction $\mathsf{tx}_i^{\mathsf{state}}$ from the given parameters. $\mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$ Returns $\mathsf{tx}_i^{\mathsf{p}}$, which takes $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ as input and creates a single $\mathsf{output} := (\alpha_i, \mathsf{OneSig}(U_{i+1}))$. $\mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i})$ Return $\mathsf{tx}_i^{\mathsf{r}}$, which takes as input $\mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0]$ and $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$. The calling user U_i makes sure that this output belongs to an address under U_i's control. It creates a single output $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\alpha_i + \epsilon, \mathsf{OneSig}(U_i))$, where α_i, U_i, U_{i+1} are taken from $\mathsf{tx}_i^{\mathsf{state}}$.

(b) 2-party operation: $\underline{\texttt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha_i', T')}$

Let T be the timeout, α_i the amount and $\theta_{\epsilon_{i-1}}$ be the output used for the two party contract set up between U_{i-1} and U_i, known from `pcSetup` executed in the Open [AMSKM21] phase. $\underline{U_i}$: $\mathsf{tx}_{i-1}^{\mathsf{state}'} := \texttt{genState}(\alpha_i', T', \overline{\gamma_{i-1}})$, $\mathsf{tx}_{i-1}^{\mathsf{r}'} := \mathsf{GenRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$, then send $(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))$ to U_{i-1} //$\theta_{\epsilon_{i-1}}$ known as θ_{ϵ_x} from `pcSetup` $\underline{U_{i-1}}$ upon $(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))$:

1. Extract α_i' and T' from $\mathsf{tx}_{i-1}^{\mathsf{state}'}$. Check that $\alpha_i' \leq \alpha_i$, $T' \geq T$ and $\mathsf{tx}_{i-1}^{\mathsf{state}'} = \texttt{genState}(\alpha_i', T', \overline{\gamma_{i-1}})$.If $U_{i-1} = U_0$, ensure that $\alpha_i' \leq x + n \cdot \mathsf{fee}$ where x is the final balance of U_n in the virtual channel. Check that $\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})$ is a correct signature of U_i for $\mathsf{tx}_{i-1}^{\mathsf{r}'}$. Check that $\mathsf{tx}_{i-1}^{\mathsf{r}'} = \mathsf{GenRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ //α_i, T and $\theta_{\epsilon_{i-1}}$ from `pcSetup`
2. $\mathsf{updateChannel}(\overline{\gamma_{i-1}}, \mathsf{tx}_{i-1}^{\mathsf{state}'})$
3. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}'}$, respectively. Return $(\top, \alpha_i', T')$. Else, return $\bot$.

$\underline{U_i}$: Upon $(\mathsf{update{-}ok})$, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$, $\mathsf{tx}_{i-1}^{\mathsf{r}}$ and $\mathsf{tx}_{i-1}^{\mathsf{p}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$, $\mathsf{tx}_{i-1}^{\mathsf{r}'}$ and $\mathtt{TX}_{\mathtt{f}}^{\mathtt{i-1}} := \mathsf{GenPay}(\mathsf{tx}_{i-1}^{\mathsf{state}'})$, respectively.

(c) Protocol:

OpenVC

(i) Setup* (see also Appendix F.5, Figure F.8), as in [AMSKM21], except:

- Create $\mathsf{tx}^{\mathsf{vc}}$ instead of $\mathsf{tx}^{\mathsf{er}}$ as shown in Figure 7.3b
- $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}(\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$ together with U_n after creating $\mathsf{tx}^{\mathsf{vc}}$, to create the VC commitment transactions.

(ii) Open and Finalize (see also Appendix F.5, Figure F.8) as in [AMSKM21]

UpdateVC

Both $U_i \in \overline{\gamma_{\mathsf{vc}}}.\mathsf{users}$ can update $\overline{\gamma_{\mathsf{vc}}}$: Create a new state $\mathsf{tx}_i^{\mathsf{state}}$ and call $\mathsf{updateChannel}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$.

CloseVC/ProlongVC (synchronized modification)

(i) InitClose/InitProlong

- $\underline{U_n}$: Let α_i' be the final balance of U_n in the virtual channel and $T' = T$ (Close) or let $T' > T$ be the new lifetime of the VC and leave $\alpha_i' = \alpha_i$ (Prolong). Execute $\texttt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha', T')$
- $\underline{U_{i-1}\text{ upon }(\top, \alpha_i', T')}$: If $U_{i-1} \neq U_0$, let $\alpha_{i-1}' := \alpha_i' + \mathsf{fee}$, exe. $\texttt{2pModify}(\overline{\gamma_{i-2}}, \mathsf{tx}^{\mathsf{vc}}, \alpha_{i-1}', T')$

(ii) Emergency-Offload: If U_0 has not successfully performed `2pModify` with the correct value α' (plus fee for each hop) until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{vc}}))$. Else, update $T := T'$

Respond (see also Appendix F.5, Figure F.8) as in [AMSKM21]

Figure 7.8: (a) macros, (b) 2-party operation, (c) protocol

a better level of privacy. We include an illustration of the full construction and the offload operation Figures F.2 and F.3 in Appendix F.3.

7.6 Security analysis

7.6.1 Informal security analysis

Balance security. When the VC is opened, a Blitz [AMSKM21] collateral payment is simultaneously opened from sender to receiver. A Blitz payment provides balance security to the intermediaries. An intermediary is merely involved in a payment, the outcome of which is atomically determined by whether or not $\mathsf{tx}^{\mathsf{vc}}$ is posted. For both of these outcomes, the intermediary does not lose money. As already argued in Section 7.4 the *synchronized modification* operation does not put an intermediary at risk.

Endpoint security. An honest sender can always enforce the VC that holds its correct balance by posting $\mathsf{tx}^{\mathsf{vc}}$ and thereby offloading the VC. By doing so, the refunding of the collateral along the path is triggered, including the one of the sender itself. This means that in case of a dispute or someone not cooperating, the sender can always use the offloading before T to ensure its balance. An honest receiver will get its rightful balance either when the channel is offloaded or, if it is not, after time T through the collateral, which is moved from left to right along the path.

Reliability. Only the sender is able to offload the VC. This means that if sender and receiver are honest, no one can force them to offload the VC before T.

Endpoint anonymity and path privacy. $\mathsf{tx}^{\mathsf{vc}}$ is constructed, as in Blitz, based on fresh and stealth addresses and the endpoints of the VC rely on fresh addresses too. Hence, an intermediary observing $\mathsf{tx}^{\mathsf{vc}}$ learns no meaningful information about the sender, the receiver, and the path. This holds only in the optimistic case. In the pessimistic case, it might be possible to link (parts of) the path to $\mathsf{tx}^{\mathsf{vc}}$ and also link the VC to sender/receiver, like in any other off-chain protocol, including the Lightning Network.

Value privacy. Similarly to how payments between users of a payment channel (PC) are known only to those users, also VC updates are only known to the endpoints. There occur no on-chain transactions in the optimistic case throughout the protocol. Any two users connected in the PC network can open a VC, and apart from their open and close balance, the amount and nature of the individual updates remains known only to them, even in the pessimistic case.

7.6.2 Security model

We rely on the synchronous, global universal composability (GUC) framework [CDPW07] to model the Donner protocol. We make use of some preliminary functionalities commonly used in the literature [AEE+21, DEF+19b, DEFM19, AMSKM21, AME+21]. The global ledger $\mathcal{L}$ is maintained by the functionality $\mathcal{G}_{Ledger}$, which is parameterized by a signature

scheme Σ and a blockchain delay Δ, i.e., an upper bound on the number of rounds it takes for a valid transaction to appear on $\mathcal{L}$, after it is posted. The notion of time (or computational rounds) is modeled by $\mathcal{G}_{clock}$ and the communication by $\mathcal{F}_{GDC}$. Finally, a functionality $\mathcal{F}_{Channel}$ handles the creation, update, and closure of PCs as well as the preparation and update of the VCs.

We define an ideal functionality $\mathcal{F}_{Pay}$ that models the idealized behavior of our VC protocol, stipulating input/output behavior, impact on the ledger as well as possible attacks by adversaries. In the ideal world, $\mathcal{F}_{Pay}$ is a trusted third party. Additionally, we formally define the real-world hybrid protocol Π and show that Π *emulates* (or realizes) $\mathcal{F}_{Pay}$. For this, we describe a simulator $\mathcal{S}$ that translates any attack of any adversary on Π into an attack on $\mathcal{F}_{Pay}$.

To show that the protocol Π realizes $\mathcal{F}_{Pay}$, we need to show that no PPT *environment* $\mathcal{E}$ can distinguish between interacting with the real world and interacting with the ideal world with non-negligible probability. This implies, that any attack that is possible on the protocol is also possible on the ideal functionality. Intuitively, it suffices to output the same messages and add the same transaction to the ledger in both the real and the ideal world in the same rounds. We refer to Appendix F.6 for the preliminaries, the ideal functionality, the formal protocol, the simulator, the formal proof of Theorem 7 and the formalization of the security and privacy goals (Definitions 21 to 26) as well as the proof that $\mathcal{F}_{Pay}$ has these properties (Theorems 19 to 24).

Theorem 7. *For functionalities $\mathcal{G}_{Ledger}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, $\mathcal{F}_{Channel}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

7.7 Evaluation and comparison

Communication overhead. We implemented a small proof-of-concept that creates the raw Bitcoin transactions necessary for Donner [Don21]. In this implementation, we use the library `python-bitcoin-utils` and Bitcoin Script to build the transactions and test their compatibility with Bitcoin by deploying them on the testnet. We show the results for the operations *Open*, *Update*, *Close*, *Offload* in Table 7.2. For transactions that go on-chain, we provide additionally the expected cost in USD at the time of writing. For this evaluation, we assume generalized channels [AEE+21] as the underlying payment channel (PC) protocol, but note that Donner is also compatible with Lightning channels (as we discuss at the end of this section).

For opening a virtual channel (VC), each of the n underlying PCs needs to exchange 4 transactions: $\mathsf{tx}^{\mathsf{vc}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and two transactions for updating the state. Since $\mathsf{tx}^{\mathsf{vc}}$ has an output for every intermediary and the sender, its size increases with the number of channels on the path n and is $192 + 34 \cdot n$ bytes. $\mathsf{tx}_i^{\mathsf{r}}$ has a size of 306 bytes, and a channel update to a state holding this contract is 742 bytes. $\mathsf{tx}_i^{\mathsf{p}}$ does not need to be exchanged, since the left user of a channel can generate it independently. This totals to $1240 + 34 \cdot n$ bytes of off-chain communication per channel for the open phase. Then, we require to

Table 7.2: Communication overhead of Donner for the whole path (not per party) for the different operations, assuming a VC across n channels. In the pessimistic offload, $k \in [0, n]$ is the number of channels where there is a dispute. Only in the Offload case transactions are posted on-chain.

	# txs	size (bytes)	on-chain cost (USD)
Open	$4 \cdot n + 2$	$34 \cdot n^2 + 1240 \cdot n + 695$	0
Update	2	695	0
Close	$3 \cdot n$	$1048 \cdot n$	0
Offload (Optimistic)	1	$192 + 34$	$0.25 + 0.04 \cdot n$
Offload (Pessimistic)	$3k + 1$	$1048 \cdot k + 192 + 34 \cdot n$	$1.36 \cdot k + 0.25 + 0.04 \cdot n$

exchange the initial state of the VC, which is 2 transactions or 695 bytes. This totals $4 \cdot n + 2$ transactions or $34 \cdot n^2 + 1240 \cdot n + 695$ bytes for the path.

For honestly closing a VC, the payment needs to be updated from right to left. However, $\mathsf{tx}^{\mathsf{vc}}$ does not need to be exchanged anymore, so we only need to exchange 3 transactions or 1048 bytes for each of the n underlying channels. To update a VC, the two endpoints need to exchange 2 transactions with 695 bytes, the same as a PC update.

Finally, for offloading, only the transaction $\mathsf{tx}^{\mathsf{vc}}$ needs to be posted on-chain and nothing per channel. This means $192 + 34 \cdot n$ bytes and costs $0.25 + 0.04 \cdot n$ USD. Note that if individual users on the path do not collaborate, regardless if the VC is offloaded or successfully closed, these channels may need to be closed as well. We argue that this is also the case during the normal PC execution, e.g., when routing multi-hop payments. However, for every channel that does need to be closed, the three transactions exchanged in the close phase need to be posted additionally. If there are k channels with such a dispute, this results in a total of $3k + 1$ transactions or $1048 \cdot k + 192 + 34 \cdot n$ bytes, which costs $1.36 \cdot k + 0.25 + 0.04 \cdot n$ USD for the whole path. We mark this as the *pessimistic* case in Table 7.2.

Efficiency comparison. We compare our construction to LVPC [JLT20] and Elmo [KL] (cf. Table 7.3), the only current Bitcoin-compatible VC solutions over multiple hops. As already mentioned, LVPC and Elmo have rooted VC funding transactions. We evaluate, in particular, the off-chain and on-chain costs of the core VC operations (open, update, close, and offload), concluding that Donner is better in each case.

LVPC is constructed recursively; there are different ways of doing the recursion. Each combination leads to the same minimum number of VCs required for a path of n base channels: One for each of the $n-1$ intermediaries. The storage overhead per intermediary is linear in the number of layers on top of a user, which in turn is in the worst case linear (Figure F.6 in Appendix F.5.1) and in the best case logarithmic (Figure F.7 in Appendix F.5.1) in the path length.

In the open phase across the whole path, Donner requires $4 \cdot n + 2$ off-chain transactions for the whole path. In LVPC, 7 off-chain transactions per VC are needed, so $7 \cdot (n-1)$. Similarly, for closing, we need to store 4 transactions per VC in LVPC, so $4 \cdot (n-1)$. Elmo requires to store $n - 2 + \chi_{i=2} + \chi_{i=n-1} + (i - 2 + \chi_{i=2})(n - i - 1 + \chi_{i=n-1}) \in \Theta(n^2)$

Table 7.3: Comparison of LVPC, Elmo and Donner for a VC over from U_0 to U_n.[1]In the pessimistic offload in Donner, $k \in [0, n]$ is the number of channels where there is a dispute.

		# txs	off-chain
Open	LVPC [JLT20]	$7 \cdot (n-1)$	✓
	Elmo [KL]	$\Theta(n^3)$	✓
	Donner	$4 \cdot n + 2$	✓
Update	LVPC [JLT20]	2	✓
	Elmo [KL]	2	✓
	Donner	2	✓
Close	LVPC [JLT20]	$4 \cdot (n-1)$	✓
	Elmo [KL]	$3 \cdot n + 3$	✗
	Donner	$3 \cdot n$	✓
Offload	LVPC [JLT20]	$5 \cdot (n-1)$	✗
(Optimistic)	Elmo [KL]	$3 \cdot n + 1$	✗
	Donner	1	✗
Offload	LVPC [JLT20]	$5 \cdot (n-1)$	✗
(Pessimistic)	Elmo [KL]	$3 \cdot n + 1$	✗
	Donner[1]	$3 \cdot k + 1$	✗

(where χ_P is 1 if P is true and 0 otherwise) for the i^{th} intermediary (and 1 for the endpoints), resulting in a storage overhead of $\Theta(n^3)$ for the whole path. Closing honestly (i.e., off-chain) is not defined for Elmo, so it needs to be closed on-chain, resulting in 2 transactions per channel (n) for closing plus 1 transaction per user ($n+1$) plus 2 transactions to close the VC or $3 \cdot n + 3$ on-chain transactions. Donner requires the close operation per underlying channel, so $3 \cdot n$ transactions. The update phase is the same in all constructions.

The interesting case again is the offload case. As we already pointed out, a fully rooted, recursive VC construction requires to close *all underlying channels*. This means in LVPC, we require 2 transactions per underlying channel, of which we have n PCs and $n-2$ VCs (all but the topmost one). Additionally, we need to publish $n-1$ funding transactions of the VCs including the topmost one. This results in $2 \cdot (2n-2) + n - 1 = 5 \cdot n - 5$ transactions that have to be posted on-chain along with the fact that all involved channels have to be closed in the case of a dispute. In Elmo, we need $3 \cdot n + 1$, i.e., the number of transactions to close minus the 2 transactions required to put the VC state on-chain. In Donner, only 1 transaction has to be posted on-chain. For the pessimistic offload, there need to be $3 \cdot k + 1$ transactions posted in Donner, where k is the number of channels

where there is a dispute. We show an application example in Appendix F.1.1, where we analyze how Donner can be used to connect a node better to a network via VCs, compared to no VCs and LVPC.

Compatibility with LN channels. To simplify the formalization of this work, we built our VC construction on top of generalized payment channels (GC) [AEE+21], which have one symmetric channel state. However, it is also possible to construct Donner on top of LN channels, which have two asymmetric channel states. The (one-hop) BCVC [AME+21] constructions rely on GCs as well, while the recursive LVPC [JLT20] relies on simple channels that have only one state, but each update reduces the limited lifetime of the channel. (Elmo [KL] needs the opcode `ANYPREVOUT` that is not supported in Bitcoin or in the LN.)

As LN channels are the only ones deployed in practice so far, it is interesting to investigate the effect of building VCs on top of LN channels. We point out that building Donner on top of LN channels is not difficult, as the collateralization in the underlying base channels is similar to a MHP. In fact, the only two differences for implementing Donner on top of LN channels instead of GCs is that (i) for each of the two asymmetric states per channel we now need to create a tx_i^r transaction, so two instead of one, and (ii) a punishment mechanism has to be introduced per output instead of per state (e.g., similar to how HTLCs are handled in LN).

The LVPC construction is not as straightforward to implement on top of LN channels. Similarly to Donner, we need to introduce a punishment mechanism (ii). However, the more difficult part is handling the two asymmetric states (i). Since the VC needs to be able to be posted regardless of which of the two states are posted, there needs to be a unique funding transaction (called Merge in [JLT20]) for each possible combination of states in the underlying channels. This implies that in a LVPC-like construction which is built on top of LN channels, the storage overhead per party is *exponential* in the layers of VCs that are constructed over this party. In fact, using channels with duplicated states this exponential growth is present in every rooted, recursive VC construction. This follows from the evaluation in [AEE+21]. For each of these exponentially many copies of the VC, commitment transactions need to be exchanged for an update, so there is an exponential communication overhead too. Note that the storage overhead for Donner on top of LN channels is *constant* as is the communication overhead for updates.

7.8 Conclusion

Payment channel networks (PCNs) have emerged as successful scaling solutions for cryptocurrencies. However, path-based protocols are tailored to payments, excluding novel and interesting non-payment applications such as Discreet Log Contracts, while creating direct PCs on-demand is expensive, slow, and infeasible on a large scale. VCs are among the most promising solutions. We show that all existing UTXO-based constructions are vulnerable to the Domino attack, which fundamentally undermines the underlying PCN itself.

Hence we introduce a new VC design, the first one to be secure against the Domino attack, besides the only one achieving path privacy and a time-based fee model. Our performance analysis demonstrates that Donner is more efficient: It only requires a single on-chain transaction to solve disputes, as opposed to a number that is linear in the path length, and the storage overhead is constant too, as opposed to linear.

Overall, Donner offers an easy-to-adopt, LN-compatible VC construction enabling new applications such as Discreet Log Contracts or fast and direct micropayments, without the need to create a direct PC. Unlike the underlying PCNs, the VCs are not susceptible to liveness and privacy attacks by the intermediaries and do not require fees per payment.

Acknowledgements. This work has been supported by the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC); by the Austrian Science Fund (FWF) through the projects PROFET (grant P31621) and the project W1255-N23; by the Austrian Research Promotion Agency (FFG) through the Bridge-1 project PR4DLT (grant 13808694) and the COMET K1 SBA and COMET K1 ABC; by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP); by CoBloX Labs; by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association through the Christian Doppler Laboratory Blockchain Technologies for the Internet of Things (CDL-BOT); by the National Science Foundation (NSF) under grant CNS-1846316; by Madrid regional government as part of the program S2018/TCS-4339 (BLOQUES-CM) co-funded by EIE Funds of the European Union; by the project HACRYPT (N00014-19-1-2292); by grant IJC2020-043391-I/MCIN/AEI/10.13039/501100011033 and European Union NextGenerationEU/PRTR; by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR; by SCUM Project (RTI2018-102043-B-I00) MCIN/AEI/10.13039/501100011033/ERDF A way of making Europe.

CHAPTER 8

Conclusion and Directions of Future Research

8.1 Conclusion

In this thesis, we have shown how protocols built on top of cryptocurrencies can be used to improve scalability. To this end, we have introduced novel protocols for payment channels, payment channel networks, and virtual channels, improving upon the state of the art in off-chain scaling solutions in terms of security, privacy, efficiency, and allowing for new applications.

First, we introduced a payment channel protocol allowing users to go offline safely. Thereby, we get rid of the online assumption, one of the major disadvantages and differences of existing layer-2 (off-chain) solutions over layer-1 (blockchain). Further, we formalize the notion of generalized channels and provide a protocol that effectively bridges the gap between payment channels and state channels. Generalized channels can be used to lift any scripting capabilities of the underlying blockchain to the off-chain setting. We eliminate state duplication, handling the punishment mechanism instead with adaptor signatures, which we formalize for the first time.

Second, we introduce a new multi-hop payment scheme that, for the first time, consolidates the efficiency of one-round payments with the security of two-round payments. This is not only theoretically interesting but also has an impact in practice, as we demonstrate in our simulation on the effect of constant collateral lock time. Moreover, we introduce a multi-channel update scheme, which allows us to securely and atomically update multiple payment channels that do not have to be aligned on a path. Instead, they can have any topology, which opens the door to new and exciting applications, such as off-chain crowdfunding or transaction aggregation.

Finally, we introduce the first Bitcoin-compatible virtual channel protocol, demonstrating that virtual channels can be built without relying on Turing-complete scripting. Further, we analyze this construction as well as concurrent virtual channel constructions for Bitcoin. We find a common design paradigm that allows for a devastating novel attack. We simulate the severity of this attack, showing that these existing constructions should not be used in practice. Moreover, we present a generic multi-hop virtual channel construction that is not only secure against this attack but also improves efficiency and allows for a fair economic fee model. Such a general virtual channel construction enables hosting any application (e.g., DLCs) in a multi-hop setting.

8.2 Directions for Future Work

Collateral. One fundamental drawback of payment channel networks as scalability solutions is that they require a large amount of money to be locked up in channels. The normal functioning of PCNs requires intermediary users to lock up some collateral, both for routing payments and for building virtual channels. To resolve this issue, there are some potential future research directions. Existing countermeasures include making channel allocation more effective, both for payment and virtual channels, e.g., [KR21, ACM+]. More interesting still would be to enable users who have money in the PCN to be able to use that money for anything while further reducing or removing the need for collateral of intermediaries. It would be interesting to explore how the advantages of PCNs and, e.g., rollups or other scalability can be combined, potentially in a hybrid approach.

Interoperability. There is a vast number of different cryptocurrencies, each with a unique set of features. To avoid using centralized services such as exchanges, it is crucial to have cross-chain protocols. In the context of PCNs, cross-chain multi-hop payments or swaps are already possible using HTLCs or adaptor signatures, assuming that there is at least one party on both chains. Exploring how one could build cross-chain virtual channels, atomic multi-channel updates, or other off-chain applications would be an exciting line of research. But not only the interoperability between different blockchains is interesting, also making different scaling solutions compatible is an exciting research direction, e.g., PCNs and rollups, sidechains, etc.

Privacy. PCNs are somewhat revealing, as they generally leak the value of each payment to intermediary users. This is necessary since the intermediary users need to lock up collateral. The situation improves somewhat through the use of virtual channels, which at least hide any operation other than opening and closing from intermediaries; in particular, they hide the number of updates and the value of each update. Still, exploring the possibility of building value-privacy-preserving multi-hop payment schemes or virtual channels that do not reveal this balance during the opening and closing of the channel would be intriguing.

Expressiveness. One of the goals of this thesis was already to explore which off-chain protocols we can build with limited scripting. An interesting follow-up would be to see

formally what the minimal required scripting capabilities of the underlying blockchain are to build a given off-chain protocol and why. For example, for what type of applications is it needed to have relative timelocks? Another compelling direction is to explore how having an account-based model instead of a UTXO-based model impacts the applications that can be expressed, given the same scripting capabilities. However, it is also interesting to see if off-chain protocols can be used to add functionality, possibly also by adding trust assumptions, such as a trusted price oracle.

Efficiency. To reduce cost and make off-chain protocols more usable in practice, minimizing the on-chain load in the pessimistic case is essential. One of the techniques we use in this thesis is to make use of auxiliary outputs, which hold a negligible amount ϵ. These outputs exploit the UTXO model and are necessary to make some transactions depend on another transaction. However, they have some impact on practicality. First, one protocol party (e.g., the sender of a payment in Blitz) needs to fund these outputs, which is undesirable even for tiny amounts. Second, these outputs are not used in the optimistic case, possibly remaining unclaimed for a long time or forever. This leads to an unnecessary load on the UTXO set, i.e., the data structure that miners use to keep unspent outputs in memory. It would be exciting to get rid of these auxiliary outputs.

List of Figures

1.1 Payment channel: A payment channel consists of three operations. Alice and Bob can (1) open a payment channel by creating a *funding transaction*, which locks some of their money in a shared account (multisig address). In this example, Alice puts 6 coins and Bob 4 coins into the funding transaction. They create another transaction (state), which pays them back their coins in this initial balance distribution $(A:6, B:4)$ and then post the funding transaction on the blockchain. Then, Alice and Bob can (2) update their channel from $(A:6, B:4)$ to, e.g., $(A:3, B:7)$, which represents Alice paying 3 coins to Bob. They can continue updating their channel as often as they want. When they are done, they finally (3) close their channel with the latest state, unlocking their coins. Only two transactions go on-chain. . . 4

1.2 Multi-hop payment (MHP): Sender A pays 4 coins to receiver D via B and C. The colored boxes connecting two users represent payment channels (as shown in Figure 1.1). The first row shows the initial balance distribution of the channels. The second row shows the desired outcome after the payment, where 4 coins were transferred to the receiver. The challenge of MHPs is to update these channels atomically in order to prevent honest users from losing their coins. 5

1.3 Virtual Channel (VC): A VC is not funded on-chain, but built on top of payment channels. For this, some funds of these underlying payment channels are locked (illustrated in gray) as collateral, such that the whole VC capacity is covered (in this example, 5 coins). This collateral is used to compensate honest users in case of misbehavior. After successfully closing a VC, its latest balance will be reflected in the underlying payment channels. All operations happen off-chain and while the VC is open, the two endpoints, A and D, can transact without the involvement of the intermediaries B and C. 6

1.4 Lightning channel: In Lightning channels, there exist two versions of each state (*state duplication*), one for Alice (e.g., State_A^0) and one for Bob (e.g., State_B^0). When updating to a new state, Alice and Bob sample new revocation secrets r_A^1 and r_B^1 uniformly at random, respectively. Then, they create the two versions of the new state State_A^1 and State_B^1, exchange the signatures of the respective state (Alice signs State_B^1, Bob signs State_B^1), and then exchange the revocation secrets of the previous states r_A^0 and r_B^0. If Bob now tries to cheat by posting State_B^0 because he holds more coins than in State_B^1, there is a spending condition on his balance, and he cannot spend it right away. Alice has one day to use r_B^0 to punish Bob and steal all his money. If Bob had posted the latest state State_B^1, Alice would not have known r_B^1 yet and could not have punished Bob. This ensures that honest parties cannot lose their funds, but comes at the cost of users constantly needing to monitor the blockchain to react to misbehavior, having two versions of each state, and needing a punishment mechanism per output. 8

1.5 Hash Time Locked Contract (HTLC): This example shows an HTLC inside the channel of Alice and Bob. Here, Alice and Bob create and update their payment channel to a new state, where Alice locks 4 of her coins in an HTLC, conditioned on y and $t = 2\text{days}$. These 4 coins can be spent either by Bob, if Bob knows a pre-image r, such that $\mathcal{H}(r) = y$, or else by Alice, after 2days. 10

1.6 Lightning payment: Sender A pays 4 coins to receiver D via B and C. The colored boxes connecting two users represent payment channels (as shown in Figures 1.1 and 1.5). The first row shows the initial balance distribution of the channels. The last row shows the desired outcome, where 4 coins were transferred to the receiver. These updates should occur atomically. In the case of Lightning-based HTLC payments, this is achieved in four steps. (1) D samples a uniformly random string r, and (2) sends its hash $y := \mathcal{H}(r)$ to A. Then, in step (3), parties lock 4 coins in an HTLC, sequentially from left to right. B and C each charge 0.1 fee in this example, thus forward only 3.8 and 3.9 coins, respectively. In step (4), after creating all HTLCs, D knowing r can unlock his HTLC with C, claiming the coins and revealing r to C, who can continue in the same way until all HTLCs are unlocked. Due to the increasing timelocks (1, 2, and 3 days, respectively), each user has enough time to propagate r. Users are incentivized to unlock the HTLC. If something goes wrong before step (3) is completed, or if D chooses not to reveal r, the HTLCs are reverted after the timelocks expire. Thus, the payment is atomic. 11

2.1 The transaction flow of LN channel between A and B. Rounded boxes represent transactions, rectangles within represent outputs of the transaction: here $v_A + v_B = f$. Incoming arrows represent transaction inputs, while outgoing arrows represent how an output can be spent. Double lines from transaction outputs indicate the output is a shared address. A single line from the transaction output indicates that the output is a single party address. We write the timelock (Δ) associated with a transaction over the corresponding arrow. 24

2.2 Transaction flow of our base solution. Here double lines from transaction outputs indicate that the output is a 2-party shared address between A and B. A single line from the transaction output indicates that the output is a single party address. We have $v_A + v_B = f$ and ϵ is some negligible amount of coins. 26

2.3 Transaction flow of the extension to our protocol. Again, $v_A + v_B = f$ and ϵ is some negligible amount of coins. The collateral c can be chosen as a value $0 \leq c \leq f$. For $c = 0$, we get Figure 2.2. 27

2.4 Ideal Functionality 31

2.5 Sleepy Channel protocol - Payment setup, payments, closing, and punishment . 36

2.6 Results of the first simulation. 40

2.7 Results of the second simulation. (Blue = LN, Red = Sleepy Channels) . 41

3.1 (Left) tx is published on the blockchain. The output of value x_1 can be spent by a transaction containing a preimage of h and signed w.r.t. pk_A. The output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t rounds passed since tx was accepted by the blockchain. (Right) tx' is not published yet. Its only output can be spent by a transaction whose witness satisfies $\varphi_1 \vee \varphi_2 \vee \varphi_3$. 50

3.2 A Lightning style payment channel where A has x_A coins and B has x_B coins. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ upper bounds the time needed to publish a transaction on a blockchain. 51

3.3 A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. In the figure, pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain. 53

3.4 The ideal functionality $\mathcal{F}_L^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T_p, k)$. We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$. 58

3.5 ECDSA-based adaptor signature scheme. 63

3.6 Schematic description of the generalized channel creation protocol. . . . 65

4.1 (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx' has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown. 78
4.2 Illustration of the pay-or-revoke paradigm. 84
4.3 Transaction $\mathsf{tx}^{\mathsf{er}}$, which enables the refunds and, here, spends the output of some other transaction $\mathsf{tx}^{\mathsf{in}}$. 84
4.4 Payment setup in the channel $\overline{\gamma_i}$ of two neighboring users U_i and U_{i+1} with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{U_i} and $x_{U_{i+1}}$ are the amounts that U_i and U_{i+1} own in the state prior to $\mathsf{tx}^{\mathsf{state}}$. 84
4.5 Protocol for 2-party channel update 91
4.6 The Blitz payment protocol 93
4.7 Subprocedures used in the protocol 94
4.8 Ratio $\mathsf{fail}_{\mathsf{LN}}/\mathsf{fail}_{\mathsf{Blitz}}$. (Left) we fix the number of disrupted payments at 0.5% and vary ub. (Right) we fix ub at 3000 and vary the number of disrupted payments. 97

5.1 An example of a payment in LN from **A** to **D** for a value α using HTLC contracts. An HTLC contract denoted by HTLC(Alice, Bob, x, y, t), shows the following conditions: (i) If timeout t expires, Alice gets back the locked x coins. (ii) If Bob reveals a value r, such that $\mathcal{H}(r) = y$, before timeout t, Alice pays x coins to Bob. 103
5.2 The left transaction tx has two outputs, one of value x_1 that can be spent by A, with a transaction signed w.r.t. pk_A, but only if at least t_1 rounds passed since tx is accepted on the blockchain. The other output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B at or after round t_2. The right transaction tx' has one input, which is the second output of tx containing x_2 coins, and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee (\phi_2 \wedge \phi_3)$. The inputs of tx are not shown. 107
5.3 Update contract for the channel γ_i between two neighboring users $\gamma_i.\mathsf{sender}$ and $\gamma_i.\mathsf{receiver}$ with the new state $\mathsf{tx}^{\mathsf{state}}$. x_{S_i} is the amount that $S_i = \gamma_i.\mathsf{sender}$ owns and x_{R_i} is the amount that $R_i = \gamma_i.\mathsf{receiver}$ owns in the state before $\mathsf{tx}^{\mathsf{state}}$. 111
5.4 Transaction $\mathsf{tx}_i^{\mathsf{ep}}$ created by receiver R_i for a payment with n channels, where the set of all senders is $\{S_j\}_{j\in[1,n]}$ and the set of all receivers is $\{R_j\}_{j\in[1,n]}$. This transaction enables all payments and spends the output of transaction $\mathsf{tx}_i^{\mathsf{in}}$. 113

5.5 For each channel, first, the receiver sends her own $\mathsf{tx}^{\mathsf{ep}}$ to all other parties (the Pre-Setup message). The sender creates $\mathsf{tx}^{\mathsf{state}}$ and one tx^{p} for each $\mathsf{tx}^{\mathsf{ep}}$, then sends all these transactions to the receiver (Setup message). After verifying the message, the receiver sends her first endorsement to all other parties. When the sender gets all endorsements, she sends her signature to each $\mathsf{tx}^{\mathsf{ep}}$ to its creator (Confirmation message). After getting all signatures and verifying them, the receiver sends the second endorsement to all other parties. Finally, when the receiver has enough signatures as her $\mathsf{tx}^{\mathsf{ep}}$ witnesses, and the payment is not received, she will post her $\mathsf{tx}^{\mathsf{ep}}$ to the ledger. . . . 115

5.6 Per-channel off-chain storage overhead for varying number of synchronized channels. . . . 126

5.7 An example of rebalancing with 4 users and 5 channels. Each user holds the same coins after the rebalancing as before, but distribution of coins through channels is changed in order to refund depleted channels. In this case, rebalancing cannot be conducted using a single path-formed payment without using a channel more than once. . . . 127

6.1 (Left) Transaction tx is published on the blockchain. The output of value x_1 can be spent by a transaction signed w.r.t. pk_B after round t_2, and the output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_3 rounds passed since tx was accepted by the blockchain. (Right) Transaction tx' is not published on the ledger. Its only output, which is of value x, can be spent by a transaction whose witness satisfies the output condition $\varphi_1 \vee \varphi_2 \vee \varphi_3$. . . . 137

6.2 A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. The value of Δ upper bounds the time needed to publish a transaction on a blockchain. The condition ϱ_A represents the verification of A' revocation secret and ϱ_B represents the verification of B' revocation secret. . . . 138

6.3 A virtual channel γ built over ledger channels α, β. . . . 139

6.4 Modular creation procedure of a virtual channel on top of two ledger channels α and β. . . . 144

6.5 Modular close procedure of a virtual channel on top of two ledger channels α and β. For $P \in \{A, B\}$, $\overrightarrow{\theta}_P := \{(c_P, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})\}$ where $\gamma.\mathsf{st} = \Big((c_P, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P}), (c_Q, \mathtt{One\text{–}Sig}_{\mathsf{pk}_Q})\Big)$. . . . 146

6.6 Funding of a virtual channel γ without validity. T upper bounds the number of off-chain communication rounds between two parties for any operation in the ledger channel. . . . 147

6.7 Transactions published after a successful offload. . . . 148

6.8 Transactions published after A successfully executed the punishment procedure. The grayed transaction $\mathtt{TX}^{\mathsf{B}}_{\mathsf{s}}$ indicates that this transaction has not been published. . . . 149

6.9 A Lightning style payment channel where A has x_A coins and B has x_B coins. Δ upper bounds the time needed to publish a transaction on a blockchain. condition ϱ_A represents the verification of A' revocation secret and h represents the verification of B' revocation secret. 150
6.10 Pictorial illustration of Table 6.3. 155

7.1 Conceptual comparison of (a) state-of-the-art VCs (rooted) and (b) our protocol (non-rooted) on layers L1 (blockchain), L2 (PCs) and L3 (VCs). Note that the VC in (a.1) is funded by all the underlying channels In (b.1), the VC is funded only by U_0, indirectly via a transaction $\mathsf{tx}^{\mathsf{vc}}$. Additionally, in (b.1), a payment is set up from U_0 to U_3, whose outcome depends on whether the VC is offloaded. Offloading, i.e., the act of forcefully transforming a VC (L3) to a PC (L2) in (a.2), requires that all the underlying PCs (L2) are put on-chain (L1). In (b.2), offloading the VC keeps the PCs open, posting only $\mathsf{tx}^{\mathsf{vc}}$ on-chain (L1). Since offloading enables U_3 to receive their funds, the payment is refunded then. However, since in (b), only U_0 can offload, U_3 is compensated (b.3) after a timeout T via a payment that is executed iff U_0 has not offloaded the VC (i.e., (b.2) did not happen). 164
7.2 We abstract PCs using a squiggly line to hide details that are not needed in this work. $P : x_P$ indicates that user P owns x_P coins in the state tx^{s}, written as (A, B). The box containing $x_A + x_B$ indicates the shared output of A and B. 166
7.3 (7.3a) Illustration of the Blitz synchronization protocol; (7.3b) Off-chain synchronization transaction spending from an output under U_0's control and linking to the collateral in each channel. (i) Without the green part: $\mathsf{tx}^{\mathsf{er}}$ in Blitz. (ii) With the green part: $\mathsf{tx}^{\mathsf{vc}}$ used for funding the VC in this work; (7.3c) Two-party contract used within each channel 168
7.4 Illustration of a VC construction over a single intermediary. The VC funding tx^{f} is rooted in the underlying channels is the only way for the intermediary to get its collateral back. tx^{f} and the the punishment are mutually exclusive. 168
7.5 Illustration of a rooted VC via multiple hops. The yellow lines indicate how the VC is rooted. All transactions connected to and to the left of tx^{f} need to be put on-chain in case the rightmost VC is offloaded. 170
7.6 Simulated effect of the Domino attack. 173
7.7 Synchronized modification: Safely modify the contract tied to a transaction $\mathsf{tx}^{\mathsf{vc}}$ in each channel atomically. Note that $\mathsf{tx}^{\mathsf{vc}}$ is the same transaction in all three cases. 177
7.8 (a) macros, (b) 2-party operation, (c) protocol 182

B.1 Description of the global ledger functionality. 239
B.2 Schnorr-based adaptor signature scheme $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$. 241
B.3 The formal definition of game $\boldsymbol{G_0}$. 244
B.4 The formal definition of $\boldsymbol{G_1}$. 244
B.5 The formal definition of $\boldsymbol{G_2}$. 245

B.6 The formal definition of $\boldsymbol{G_3}$. 246
B.7 The formal definition of $\boldsymbol{G_4}$. 247
B.8 The formal definition of the simulator. 248
B.9 The formal definition of $\boldsymbol{G_0}$. 250
B.10 The formal definition of game $\boldsymbol{G_0}$. 250
B.11 The formal definition of game $\boldsymbol{G_2}$. 251
B.12 The formal definition of game $\boldsymbol{G_3}$. 252
B.13 The formal definition of game $\boldsymbol{G_4}$. 253
B.14 The formal definition of the simulator. 254
B.15 The formal definition of game $\boldsymbol{G_0}$. 257
B.16 The formal definition of game $\boldsymbol{G_1}$. 258
B.17 The formal definition of game $\boldsymbol{G_2}$. 258
B.18 The formal definition of game $\boldsymbol{G_3}$. 259
B.19 The formal definition of game $\boldsymbol{G_4}$. 260
B.20 The formal definition of the simulator. 261
B.21 The formal definition of the game $\boldsymbol{G_0}$. 263
B.22 The formal definition of the game $\boldsymbol{G_1}$. 263
B.23 The formal definition of the game $\boldsymbol{G_2}$. 264
B.24 The formal definition of the game $\boldsymbol{G_3}$. 265
B.25 The formal definition of the game $\boldsymbol{G_4}$. 266
B.26 The formal definition of the game $\boldsymbol{G_3}$. 267

C.1 Concurrent payments between users U_i and U_{i+1}: (left) a Blitz channel with a single payment; (right) an updated channel that has this payment and a second concurrent one. To add a second payment of value α_i' to the channel, the transactions for the in-flight payment of value α_i are recreated with the new state $\mathsf{tx}_i^{\mathsf{state}\prime}$ as input, the channel is updated to $\mathsf{tx}_i^{\mathsf{state}\prime}$ and finally, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked. In the LN, this process is the same, except that the HTLC contract and transactions are recreated, instead of the Blitz ones. 289
C.2 Timeline of when transactions appear on the ledger $\mathcal{L}$ in the case payment and refund. $\tau_n - \tau_0$ denotes the time needed for the setup of the whole payment. 291

E.1 The root sets of transaction tx_8 are $\{\mathsf{tx}_1\}$, $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4\}$, $\{\mathsf{tx}_5, \mathsf{tx}_6\}$, $\{\mathsf{tx}_4, \mathsf{tx}_5\}$ and $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_6\}$. 364
E.2 Funding of a virtual channel γ with validity $\gamma.\mathsf{val}$. 368

F.1 Illustration showing the transactions that go on-chain in case of offloading, an operation that can be forced by a malicious enduser in the Domino attack, forcing all underlying channels to be closed. 408
F.2 Illustration of a Donner VC of U_0 and U_4 via U_1, U_2 and U_3. 408
F.3 Illustration of the offload operation for a Donner VC. Note that the underlying PCs remain open and only one transaction goes on-chain: $\mathsf{tx}^{\mathsf{vc}}$. 409

F.4 (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx' has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown. 409
F.5 Example of an MHP in a PCN. Here, U_0 pays 4 coins (disregarding any fees) to U_4, via U_1, U_2 and U_3. The lines represent payment channels. We write balances as (x, y), where x is the balance of the user on the right, and y the balance of the user on the left. Above we write the channel balances before and below after the payment. In an MHP, this change of balance should happen atomically in every channel (or not at all). 410
F.6 Recursive virtual channel: Example A 413
F.7 Recursive virtual channel: Example B 413
F.8 Pseudocode of the protocol. 415
F.9 Protocol for 2-party channel update. 416
F.10 Interface of $\mathcal{F}_{Channel}(T, k)$. 418

List of Tables

2.1 Comparison among payment channel approaches. We do not consider [AKKWZ21, CCF+21] as they rely on third-party committees with additional trust assumptions. Online assumption refers to the honest user being online for revocation of an old state on-chain. Unrestricted lifetime means the protocol does not require users to close the channel before a pre-specified time. Unbounded payments refer to channel users making any number of payments while the channel is open. In terms of scripts, DS refers to digital signatures, SIGHASH_NOINPUT refers to a specific signature scheme [DRO], Seq. number refers to attaching a state number to a transaction and verifying if it is greater or smaller than the current height of the blockchain. In the case of Duplex [DW15], d is the number of payments made in the channel. LRS refers to the Linkable Ring Signature scheme used in Monero [TMSS22], and DLSAG refers to the transaction scheme proposed in [MSBL+20]. 20

3.1 Costs of lightning (LC) and generalized channels (GC) funding m HTLCs. 70

4.1 Features of different payment methods: Interledger (ILP), Lightning Network (LN), Anonymous Multi-Hop Locks (AMHL), Blitz, and Blitz using the fast track payment (FT). We abbreviate timelocks as TL and signature functionality as σ. * The requirement of HTLC can be dropped from the LN using scriptless scripts when feasible. 87

4.2 Collateral time for the LN, AMHL and Blitz for unsuccessful (refund) and successful payments (pay) as well as different threat models. We say *instant* when no one on the path stops the payment in either round. ξ denotes the time users need to claim their funds (e.g., in the LN 144 blocks). 87

5.1 Comparing different payment methods: Lightning Network, Anonymous Multi-Hop Locks (AMHL), Sprites, Payment Trees, Atomic Multi-Channel Updates(AMCU), Blitz, and our construction. Studied features are: atomicity property, path restriction, need for Turing-complete smart contracts, size of per party collateral, and value privacy. For the latter, note that there are constructions that do not inherently leak the value transferred in individual channels, but they can only be used for applications (i.e., payments) that require the same value in all channels. 104

5.2 Asymptotic comparison of current solutions, with n being the number of channels. 126

5.3 On-chain overhead and cost comparison of LN, Blitz and Thora. n is the number of channels and $m \in [0, n]$ is the number of disputed channels. . . 126

6.1 Comparison of security and efficiency goals for ledger channels (L), virtual channels with validity (VC-V), and virtual channels without validity (VC-NV). 141
6.2 Evaluation of the virtual channels. For each operation, we show the number of on-chain and off-chain transactions (*# txs*) and their *size* in bytes. For on-chain transactions, *cost* is in USD and estimates cost of publishing them on the ledger. 153
6.3 Comparison of virtual channels (VC) to multi-hop payments (PCN) showing the overhead in bytes for a different number of payments and the difference in fees. 155

7.1 Comparison to other multi-hop VC protocols. * by synchronizing all channels, this time can be only $\Theta(\log(n))$. 162
7.2 Communication overhead of Donner for the whole path (not per party) for the different operations, assuming a VC across n channels. In the pessimistic offload, $k \in [0, n]$ is the number of channels where there is a dispute. Only in the Offload case transactions are posted on-chain. 185
7.3 Comparison of LVPC, Elmo and Donner for a VC over from U_0 to U_n.[1]In the pessimistic offload in Donner, $k \in [0, n]$ is the number of channels where there is a dispute. 186

A.1 Overhead for operations, given a current fee of 102 satoshi per byte and a price of 57, 202 USD per BTC. 234

C.1 Communication overhead of the LN and Blitz. The pessimistic transactions are on-chain, the rest off-chain. 293
C.2 Extended results of our simulation. 294
C.3 Explanation of the sequence names used in Lemma 17 and where they can be found in the ideal functionality (IF), Protocol (Prot) or Simulator (Sim). 312
C.4 Explanation of the sequence names used in Lemma 18 and where they can be found. 314
C.5 Explanation of the sequence names used in Lemma 19 and where they can be found. 315

F.1 Bootstrapping cost comparison 405
F.2 Comparison to other virtual channel protocols. We denote *dispute* as the case where a party needs to enforce their VC funds or be compensated. In the UTXO case, this means offloading. * by synchronizing all channels, this time can be reduced to $\Theta(log(n))$. † for single-hop constructions n is constant, however, since the action/storage overhead/time delay is per user, we write $\Theta(n)$. ‡ This depends on using indirect/direct dispute. 405

F.3 Explanation of the sequence names used in Lemma 26 and where they can be found in the ideal functionality (IF), Protocol (Prot) or Simulator (Sim). 439
F.4 Explanation of the sequence names used in Lemma 27 and where they can be found. 441
F.5 Explanation of the sequence names used in Lemma 29 and where they can be found. 443
F.6 Explanation of the sequence names used in Lemma 31 and where they can be found. 445

Bibliography

[AAM22] Lukas Aumayr, Kasra Abbaszadeh, and Matteo Maffei. Thora: Atomic and Privacy-Preserving Multi-Channel Updates. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 165–178. Association for Computing Machinery, 2022.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. Association for Computing Machinery, 2018.

[ACM+] Lukas Aumayr, Esra Ceylan, Matteo Maffei, Pedro Moreno-Sanchez, Iosif Salem, and Stefan Schmid. Optimizing Virtual Payment Channel Establishment in the Face of On-Path Adversaries. Under submission.

[ADMM16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[AEE+21] Lukas Aumayr, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures. In *Advances in Cryptology – ASIACRYPT 2021*, pages 635–664. Springer International Publishing, 2021.

[AGP98] Maha Abdallah, Rachid Guerraoui, and Philippe Pucheral. One-phase commit: does it make sense? *Proceedings 1998 International Conference on Parallel and Distributed Systems*, pages 182–192, 1998.

[AhC95] Yousef J. Al-houmaily and Panos K. Chrysanthis. Two-Phase Commit in Gigabit-Networked Distributed Databases. In *Parallel and Distributed Computing Systems*, 1995.

[AHC04] Yousef J Al-Houmaily and Panos K Chrysanthis. 1-2PC: the one-two phase atomic commit protocol. In *Symposium on Applied Computing*, 2004.

[AKKWZ21] Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous incentive-compatible payment channels. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*, page 209–230, Berlin, Heidelberg, 2021. Springer-Verlag.

[ALS+18] Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. Towards secure and efficient payment channels, 2018.

[AME+21] Lukas Aumayr, Matteo Maffei, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. Bitcoin-Compatible Virtual Channels. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 901–918. IEEE Computer Society, 2021.

[AMSKM21] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure Multi-Hop Payments Without Two-Phase Commits. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4043–4060. USENIX Association, 2021.

[AMSKM23] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Breaking and Fixing Virtual Channels: Domino Attack and Donner. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023*. The Internet Society, 2023.

[Ant14] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media, Inc., 1st edition, 2014.

[APS+22] Zeta Avarikioti, Krzysztof Pietrzak, Iosif Salem, Stefan Schmid, Samarth Tiwari, and Michelle Yeo. Hide & seek: Privacy-preserving rebalancing on payment channel networks. In *Financial Cryptography and Data Security*, 2022.

[ATLW20] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. Cerberus channels: Incentivizing watchtowers for bitcoin. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*, page 346–366, Berlin, Heidelberg, 2020. Springer-Verlag.

[ATM+22] Lukas Aumayr, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. Sleepy Channels: Bi-directional Payment Channels without Watchtowers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 179–192. Association for Computing Machinery, 2022.

[Ato22] Atomic multi-path payments (amp), 2022. https://docs.lightning.engineering/lightning-network-tools/lnd/amp.

[BCD+14] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling Blockchain Innovations with Pegged Sidechains, 2014.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE SP*, pages 459–474, 2014.

[BDM16] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *ESORICS*, pages 261–280, 2016.

[BDN18] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II*, page 435–464, Berlin, Heidelberg, 2018. Springer-Verlag.

[BDW17] Conrad Burchert, Christian Decker, and Roger Wattenhofer. Scalable funding of bitcoin micropayment channel networks. In *Stabilization, Safety, and Security of Distributed Systems*, pages 361–377, 2017.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 416–432, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[BH04] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *Information Security*, pages 61–72, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[BHK+20] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. Combining GHOST and casper. *CoRR*, abs/2003.03052, 2020.

[Bit18] Bitcoin wiki: Payment channels, 2018. https://en.bitcoin.it/wiki/Payment_channels.

[Bit20] Bitcoin-compatible virtual channels: Github repository, 2020. https://github.com/utxo-virtual-channels/vc.

[Bit22a] Bitcoin avg. transaction fee historical chart, January 2022. https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

[Bit22b] Bitcoin price in usd, feb 2022. https://coinmarketcap.com/.

[Bit22c] Bitcoin rich list, January 2022. https://bitinfocharts.com/top-100-richest-bitcoin-addresses.html.

[Bit22d] Bitcoin transaction fee estimator, average fee per byte, feb 2022. https://privacypros.io/tools/bitcoin-fee-estimator/.

[BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, pages 421–439, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BKM17] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *ASIACRYPT*, pages 410–440, 2017.

[Bli20] Blitz simulation: Github repository, 2020. https://github.com/blitz-payments/simulation.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies*, 2017.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO*, 2017.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX*, 2018.

[BNT20] Vivek Bagaria, Joachim Neu, and David Tse. Boomerang: Redundancy Improves Latency and Throughput in Payment-Channel Networks. In *FC*, 2020.

[BSA+17] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *CoRR*, abs/1711.03936, 2017.

[But21] Vitalik Buterin. An incomplete guide to rollups, January 2021. https://web.archive.org/web/20230524091432/https://vitalik.ca/general/2021/01/05/rollup.html#how-much-scaling-do-rollups-give-you.

[BZ18] Massimo Bartoletti and Roberto Zunino. Bitml: A calculus for bitcoin smart contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *CCS*, pages 83–100, 2018.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13, January 2000.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *FOCS*, 2001.

[CCF+21] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gaži, Philipp Kant, Aggelos Kiayias, and Alexander Russell. Fast isomorphic state channels. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II*, page 339–358, Berlin, Heidelberg, 2021. Springer-Verlag.

[CCM+20] Manuel MT Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *FC*, pages 525–539. Springer, 2020.

[CCX+19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains: (a position paper). In *Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally Composable Security with Global Setup. In *TCC*, 2007.

[Chi] Chia network faq. https://www.chia.net/faq/.

[CL05] Jan Camenisch and Anna Lysyanskaya. A Formal Treatment of Onion Routing. In *CRYPTO*, 2005.

[Cou23] Bitcoin Mining Council. Global bitcoin mining data review q4 2022, January 2023. https://web.archive.org/web/20240101174009/https://bitcoinminingcouncil.com/wp-content/uploads/2023/01/BMC-Q4-2022-Presentation.pdf.

[Cry] Cryptographic frontier, open problems in ethereum research. https://sites.google.com/view/cryptofrontier21.

[CZK+19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Xiaodong Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE EuroS&P*, pages 185–200, 2019.

[DDM+18] Dominic Deuber, Nico Döttling, Bernardo Magri, Giulio Malavolta, and Sri Aravinda Krishnan Thyagarajan. Minting mechanisms for blockchain – or – moving from cryptoassets to cryptocurrencies. Cryptology ePrint Archive, Report 2018/1110, 2018. https://eprint.iacr.org/2018/1110.

[DEF+19a] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: Practical smart contracts on bitcoin. In *USENIX Security*, pages 801–818, 2019.

[DEF+19b] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. Multi-party Virtual State Channels. In *EUROCRYPT*, 2019.

[DEFM19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *IEEE SP*, pages 106–123, 2019.

[dev] LN developers. Bolt #2: Peer protocol for channel management. https://github.com/lightning/bolts/blob/master/02-peer-protocol.md.

[DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General State Channel Networks. In *CCS*, 2018.

[DFKP15] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology – CRYPTO 2015*, page 585–605, Berlin, Heidelberg, 2015. Springer-Verlag.

[DG09] G. Danezis and I. Goldberg. Sphinx: A Compact and Provably Secure Mix Format. In *IEEE S&P*, 2009.

[DLC21] DLC over Lightning, 2021. Available at https://mailmanlists.org/pipermail/dlc-dev/2021-November/000091.html.

[Don21] Donner vc evaluation of the communication overhead, 2021. https://github.com/donner-vc/overhead.

[DRO] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. https://blockstream.com/eltoo.pdf.

[Dry17] Thaddeus Dryja. Discreet Log Contracts, 2017. Available at https://adiabat.github.io/dlc.pdf.

[DTZG22] Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad. Twilight: A differentially private payment channel network. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 555–570, Boston, MA, August 2022. USENIX Association.

[DW15] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings*, volume 9212 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015.

[EEE20] Muhammed F. Esgin, Oğuzhan Ersoy, and Zekeriya Erkin. Post-quantum adaptor signatures and payment channel networks. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS*, pages 378–397, 2020.

[EFH+21] Andreas Erwig, Sebastian Faust, Kristina Hostáková, Monosij Maitra, and Siavash Riahi. Two-party adaptor signatures from identification schemes. In *IACR International Conference on Public-Key Cryptography*, pages 451–480. Springer, 2021.

[EFHR20] Lisa Eckey, Sebastian Faust, Kristina Hostáková, and Stefanie Roos. Splitting payments locally while routing interdimensionally. ePrint Archive, 2020. https://eprint.iacr.org/2020/555.

[Eme] Emelyanenkok (pseudonym). payment channel congestion via spam-attack. https://github.com/lightningnetwork/lightning-rfc/issues/182.

[EMSM19] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *CCS*, 2019.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 152–168, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Fou19] Lloyd Fournier. One-time verifiably encrypted signatures a.k.a. adaptor signatures, Oct 2019. https://tinyurl.com/y4qxopxp.

[Fun] Funding transaction of our evaluation on the Bitcoin testnet. https://tinyurl.com/589xku8w.

[Gita] Github repository of our Sleepy Channels evaluation. https://github.com/sleepy-channels/overhead.

[Gitb] Github repository of our sleepy channels simulation. https://github.com/sleepy-channels/simulation.

[GJKR99] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, page 295–310, Berlin, Heidelberg, 1999. Springer-Verlag.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310. Springer, 2015.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on computing*, 1988.

[GMSR+20] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. SoK: Layer-Two Blockchain Protocols. In *Financial Cryptography and Data Security*, 2020.

[Gol06] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

[GW17] Rachid Guerraoui and Jingjing Wang. How Fast can a Distributed Transaction Commit? In *PODS*, 2017.

[HAB+17] Ethan Heilman, Leen AlShenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *NDSS*, 2017.

[HSL19] Maurice Herlihy, Liuba Shrira, and Barbara Liskov. Cross-chain Deals and Adversarial Commerce. *VLDB*, 2019.

[HZ20] Jona Harris and Aviv Zohar. Flood & loot: A systemic attack on the lightning network. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, 2020.

[JKLT19] Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. Sok: A taxonomy for layer-2 scalability related protocols for cryptocurrencies. Cryptology ePrint Archive, Report 2019/352, 2019. https://eprint.iacr.org/2019/352.

[JLT20] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Lightweight Virtual Payment Channels. In *19th International Conference on Cryptology and Network Security (CANS)*, 2020.

[JLT21] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. Payment trees: Low collateral payments for payment channel networks. In *Financial Cryptography and Data Security*, 2021.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 30–41, New York, NY, USA, 2014. Association for Computing Machinery.

[KB16] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *ACM CCS 16*, pages 418 – 429, 2016.

[KBS20] Christiane Kuhn, Martin Beck, and Thorsten Strufe. Breaking and (partially) fixing provably secure onion routing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 168–185. IEEE, 2020.

[KG17] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 439–453, New York, NY, USA, 2017. Association for Computing Machinery.

[KL] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. Elmo: Recursive virtual payment channels for bitcoin. `https://eprint.iacr.org/2021/747`.

[KL19] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A Composable Security Treatment of the Lightning Network. In *CSF*, 2019.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *ACM CCS*, pages 195 – 206, 2015.

[KMP16] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal security proofs for signatures from identification schemes. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 33–61, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[KMS+16] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P*, pages 839–858, 2016.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally Composable Synchronous Computation. In *TCC*, 2013.

[KNW19] Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. Outpost: A responsive lightweight watchtower. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT '19, page 31–40, New York, NY, USA, 2019. Association for Computing Machinery.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[KR21] Julia Khamis and Ori Rottenstreich. Demand-aware channel topologies for off-chain payments. In *2021 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 272–280, 2021.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[KYP+21] George Kappos, Haaroon Yousaf, Ania Piotrowska, Sanket Kanjalkar, Sergi Delgado-Segura, Andrew Miller, and Sarah Meiklejohn. An empirical analysis of privacy in the lightning network. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 167–186, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[KZF+18] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-chains: Secure, scalable off-chain payments. Cryptology ePrint Archive, Paper 2018/642, 2018. `https://eprint.iacr.org/2018/642`.

[LEPS16] Joshua Lind, Ittay Eyal, Peter R. Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. *CoRR*, abs/1612.07766, 2016.

[Liga] Lightning network. `https://lightning.network/`.

[Ligb] Lightning network specification, bolt #1: Base protocol, ping and pong messages. `https://github.com/lightningnetwork/lightning-rfc/blob/master/01-messaging.md#the-ping-and-pong-messages`.

[Lin21] Yehuda Lindell. Fast secure two-party ecdsa signing. *J. Cryptol.*, 34(4), oct 2021.

[Lin23] Robin Linus. Bitvm: Compute anything on bitcoin, December 2023. `https://bitvm.org/bitvm.pdf`.

[LN 22] Ln snapshot, January 2022. https://ln.fiatjaf.com/.

[lnc20] lnchannels. https://web.archive.org/web/20200206130901/https://ln.bigsun.xyz/, 2020.

[MBB+19a] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. Pisa: Arbitration outsourcing for state channels. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Association for Computing Machinery, 2019.

[MBB+19b] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *Financial Cryptography and Data Security*, pages 508–526, Cham, 2019. Springer International Publishing.

[MJS+14] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy*, pages 475–490, 2014.

[MMS+19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS*, 2019.

[MMSH16] Patrick Mccorry, Malte Möser, Siamak F. Shahandasti, and Feng Hao. Towards bitcoin payment networks. In *Proceedings, Part I, of the 21st Australasian Conference on Information Security and Privacy - Volume 9722*, page 57–76, Berlin, Heidelberg, 2016. Springer-Verlag.

[MMSK+17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and Privacy with Payment-Channel Networks. In *CCS*, 2017.

[MSBL+20] Pedro Moreno-Sanchez, Arthur Blue, Duc V. Le, Sarang Noether, Brandon Goodell, and Aniket Kate. Dlsag: Non-interactive refund transactions for interoperable payment channels in monero. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 325–345, Cham, 2020. Springer International Publishing.

[MSK] Pedro Moreno-Sanchez and Aniket Kate. Scriptless scripts with ecdsa. https://tinyurl.com/yxtjo47l.

[MSYS21] Arash Mirzaei, Amin Sakzad, Jiangshan Yu, and Ron Steinfeld. Fppw: A fair and privacy preserving watchtower for bitcoin. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 151–169, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. http://bitcoin.org/bitcoin.pdf.

[NFSD20] Utz Nisslmueller, Klaus-Tycho Foerster, Stefan Schmid, and Christian Decker. Toward active and passive confidentiality attacks on cryptocurrency off-chain networks. In Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp, editors, *ICISSP*, pages 7–14, 2020.

[NTT22] Joachim Neu, Ertem Nusret Tas, and David Tse. Two more attacks on proof-of-stake ghost/ethereum. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, ConsensusDay '22, page 43–52, New York, NY, USA, 2022. Association for Computing Machinery.

[Paya] Pay A star transaction of our evaluation on the Bitcoin testnet. `https://tinyurl.com/bskz7fvx`.

[Payb] Pay A transaction of our evaluation on the Bitcoin testnet. `https://tinyurl.com/2w6aebr9`.

[Payc] Pay A,B transaction of our evaluation on the Bitcoin testnet. `https://tinyurl.com/2uwn5fvb`.

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts, 2017.

[PD16] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016.

[Per20] Perun network, 2020. `https://perun.network/`.

[PKF+18] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gaži, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 480–499, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[Poe] Andrew Poelstra. Lightning in scriptless scripts. `https://tinyurl.com/mcefmph`.

[Poe17] Andrew Poelstra. Scriptless scripts. `https://tinyurl.com/ludcxyz`, May 2017.

[PS17] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *International Symposium on Distributed Computing*, 2017.

[Rai17] Update from the raiden team on development progress, announcement of raidex, February 2017. `https://tinyurl.com/z2snp9e`.

[RMKG18] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions. In *NDSS*, 2018.

[Rol] roll_up. https://web.archive.org/web/20230410170419/https://github.com/barryWhiteHat/roll_up.

[RSW96] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto, 1996.

[Rus18] Rusty Russell. [lightning-dev] splicing proposal, 2018. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-October/001434.html.

[RVMS+21] Matteo Romiti, Friedhelm Victor, Pedro Moreno-Sanchez, Peter Sebastian Nordholt, Bernhard Haslhofer, and Matteo Maffei. Cross-layer deanonymization methods in the lightning protocol. In *FC*, pages 187–204. Springer, 2021.

[SAAM23] Giulia Scaffino, Lukas Aumayr, Zeta Avarikioti, and Matteo Maffei. Glimpse: On-demand Light Client with Constant-size Storage for DeFi. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 733–750. USENIX Association, 2023.

[SC93] James W Stamos and Flaviu Cristian. Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1993.

[Sie16] David Siegel. Understanding the dao attack. https://tinyurl.com/2bzxkn7a, 2016.

[Sim22] Simulation of domino attack, 2022. https://github.com/donner-vc/simulation.

[Smi23] Corwin Smith. Optimistic rollups, June 2023. https://web.archive.org/web/20230623142748/https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/.

[Spi] Jeremy Spillman. Spillman-style payment channels. https://tinyurl.com/uwzfb2tu.

[SVR+20] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia C. Fanti, and Mohammad Alizadeh. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *NSDI*, 2020.

[Tap21] Taproot (bip 341), 2021. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki.

[TBM+20] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Giulio Malavolta, Nico Döttling, Aniket Kate, and Dominique Schröder. Verifiable timed signatures

made practical. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1733–1750, New York, NY, USA, 2020. Association for Computing Machinery.

[TGB+21] Sri Aravinda Krishnan Thyagarajan, Tiantian Gong, Adithya Bhat, Aniket Kate, and Dominique Schröder. Opensquare: Decentralized repeated modular squaring service. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3447–3464, New York, NY, USA, 2021. Association for Computing Machinery.

[Tho22] Thora payments overhead, 2022. `https://github.com/Thora-Payments/overhead`.

[TM21] Sri Aravinda Krishnan Thyagarajan and Giulio Malavolta. Lockable signatures for blockchains: Scriptless scripts for all signatures. In *IEEE S&P*, 2021.

[TMM20] Sergei Tikhomirov, Pedro Moreno-Sanchez, and Matteo Maffei. A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network. In *IEEE S&B Workshop*, 2020.

[TMSM21a] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A2l: Anonymous atomic locks for scalability in payment channel hubs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1834–1851, 2021.

[TMSM21b] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. Post-quantum adaptor signature for privacy-preserving off-chain payments. In *FC*, 2021.

[TMSS22] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Fritz Schmid, and Dominique Schröder. Verifiable timed linkable ring signatures for scalable payments for monero. In *Computer Security – ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II*, page 467–486. Springer-Verlag, 2022.

[Tod] Peter Todd. Cltv-style payment channels. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki#Payment_Channels`.

[Tra] Transcripts from coredev.tech amsterdam 2019 meeting on sighash noinput. `https://tinyurl.com/49ryfutr`.

[Tri13] Manny Trillo. Stress test prepares visanet for the most wonderful time of the year, 2013. https://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html.

[TS15] Stefan Thomas and Evan Schwartz. A protocol for interledger payments, 2015. `https://interledger.org/interledger.pdf`.

[TYA+22] Samarth Tiwari, Michelle Yeo, Zeta Avarikioti, Iosif Salem, Krzysztof Pietrzak, and Stefan Schmid. Wiser: Increasing throughput in payment channel networks with transaction aggregation. In *ACM Advances in Financial Technologies (AFT)*, 2022.

[TZS20] Saar Tochner, Aviv Zohar, and Stefan Schmid. Route hijacking and dos in off-chain networks. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT, 2020.

[Unl] Unlinkable outsourced channel monitoring. `https://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/unlinkable-outsourced-channel-monitoring/`.

[VBOM+19] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1741–1758, New York, NY, USA, 2019. Association for Computing Machinery.

[VS18] Nicolas Van Saberhagen. Cryptonote v 2.0, 2018. `https://cryptonote.org/whitepaper`.

[WSNH19] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *ACM AFT*, pages 41–61, 2019.

[YKSN19] Bin Yu, Shabnam Kasra Kermanshahi, Amin Sakzad, and Surya Nepal. Chameleon Hash Time-lock Contract for Privacy Preserving Payment Channel Networks. In *Conference on Provable Security*, 2019.

[ZABZ+21] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J. Knottenbelt. SoK: Communication Across Distributed Ledgers. In *FC*, 2021.

[ZK-] Zk-rollup. `https://web.archive.org/web/20221130233402/https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/`.

[ZMR18] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 931–948, New York, NY, USA, 2018. Association for Computing Machinery.

APPENDIX A

Appendix to Chapter 2

A.1 UC Protocol

Using the notation introduced in Section 2.4, we here give a formal version of the protocol that is augmented in a way to model it in the UC framework. More specifically, we model the environment to capture anything that happens outside of the protocol execution as well as communication model. Additionally, we replace (i) the 2-party key generation protocol Γ_{JKGen} for a signature scheme Π_{DS} with an idealized version $\mathcal{F}_{\mathsf{JKGen}}$ and (ii) the 2-party signing protocol Γ_{Sign} for a signature scheme with an idealized version $\mathcal{F}_{\mathsf{Sign}}$. Finally, we add the possibility to honestly close payment channels in a way that requires only one on-chain transaction, i.e., by creating a transaction spending from the funding transaction and giving each user their respective balance right away.

In order to improve the readability of the protocol, we exclude checks that an honest user would naturally perform, such as that parameters given from the environment are well-formed, there is an input of the fund belonging to each of the two users holding the right amount of coins, verifying that channels to be updated or closed exist, the new state is valid or that a channel to be updated or closed is not currently being updated or closed. This can be formally handled by using a protocol wrapper, that performs these checks on the messages from the environment and drops invalid ones. We refer to [AEE+21], where such a wrapper for payment channels is formally defined and use the same in this work. Similarly, for the ideal functionality we use such a wrapper as well.

Sleepy channel protocol Π

Create

Party A upon $(\texttt{CREATE}, \mathsf{id}, \gamma, \mathsf{tid}_A) \overset{t_0}{\hookleftarrow} \mathcal{E}$:

1. Generate $(\mathsf{pk}_{\mathsf{CPay},A}, \mathsf{sk}_{\mathsf{CPay},A})$, $(\mathsf{pk}_{\mathsf{pun},A}, \mathsf{sk}_{\mathsf{pun},A})$, $(\mathsf{pk}_{\mathsf{fp},A}, \mathsf{sk}_{\mathsf{fp},A})$ and $(\mathsf{pk}_{\mathsf{ffp},A}, \mathsf{sk}_{\mathsf{ffp},A})$. Let $pkey^A_{set}$ be the set of public keys of these key pairs.
2. Extract $v_{A,0}$ and $v_{B,0}$ from $\gamma.\mathsf{st}$, and $c := \gamma.c$
3. Send $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_A, pkey^A_{set}) \overset{t_0}{\hookrightarrow} B$.
4. If $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_B, pkey^B_{set}) \overset{t_0+1}{\hookleftarrow} B$, continue. Else, go idle.
5. Using $pkey^A_{set}$ and $pkey^B_{set}$, A together with B runs $\mathcal{F}_{\mathsf{JKGen}}$ to generate the following set of shared addresses: $addr_{set} := \{\mathsf{Ch}_{AB}, \mathsf{SleepyCh}_A, \mathsf{SleepyCh}_B, \mathsf{ExitCh}_A, \mathsf{ExitCh}_B, \mathsf{aux}_A, \mathsf{aux}_B\}$ which takes t_g rounds. In case of failure, abort.
6. Generate $\mathsf{tx}_f := \mathsf{tx}([\mathsf{tid}_A, \mathsf{tid}_B], [\mathsf{Ch}_{AB}], [2 \cdot c + v_{A,0} + v_{B,0}])$
7. Let $\mathsf{tx}_{set0} \leftarrow \mathtt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$
8. Let $\mathsf{sigSet}^A_0 \leftarrow \mathtt{SignTxs}^A(\mathsf{tx}_{set_0}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$
9. A generates a signature σ_{tid_A} for the output tid_A and sends $(\mathsf{createFund}, \mathsf{id}, \sigma_{\mathsf{tid}_A}) \overset{t_0+1+t_g+t_s}{\hookrightarrow} A$.
10. If $(\mathsf{createFund}, \mathsf{id}, \sigma_{\mathsf{tid}_B}) \overset{t_0+2+t_g+t_s}{\hookleftarrow} B$, post $(\mathsf{tx}_F, \{\sigma_{\mathsf{tid}_A}, \sigma_{\mathsf{tid}_B}\})$ to $\mathcal{L}$.
11. If tx_F is accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + 2 + t_g + t_s + \Delta$, store $\Gamma^A(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{set0}, \mathsf{sigSet}^A_0, addr_{set}, pkey^A_{set}, pkey^B_{set})$ and $(\mathtt{CREATED}, \mathsf{id}) \overset{t_1}{\hookrightarrow} \mathcal{E}$.

Update

Party A upon $(\mathtt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{t_0}{\hookleftarrow} \mathcal{E}$

1. $(\mathsf{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{t_0}{\hookrightarrow} B$

Party B upon $(\mathsf{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} A$

1. Retrieve $(\mathsf{tx}_F, \mathsf{tx}_{set_{i-1}}, \mathsf{sigSet}^B_{i-1}, addr_{set}, pkey^A_{set}, pkey^B_{set}) = \Gamma^B(\mathsf{id})$
2. Extract $v_{A,i}$ and $v_{B,i}$ from $\overrightarrow{\theta}$, and c from tx_F
3. Let $\mathsf{tx}_{set_i} \leftarrow \mathtt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$
4. Let $\overrightarrow{\mathsf{tid}} := (\mathsf{tx}^A_{\mathsf{Pay},i}.\mathsf{id}, \mathsf{tx}^B_{\mathsf{Pay},i}.\mathsf{id})$ be a tuple of the transaction ids of transaction $\mathsf{tx}^A_{\mathsf{Pay},i}$ and $\mathsf{tx}^B_{\mathsf{Pay},i}$.
5. $(\mathtt{UPDATE\text{–}REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \overrightarrow{\mathsf{tid}}) \overset{\tau_0}{\hookrightarrow} \mathcal{E}$
6. $(\mathsf{updateInfo}, \mathsf{id}) \overset{\tau_0}{\hookrightarrow} A$

Party A upon $(\mathsf{updateInfo}, \mathsf{id}) \overset{t_0+2}{\hookleftarrow} B$

1. Retrieve $(\mathsf{tx}_F, \mathsf{tx}_{set_{i-1}}, \mathsf{sigSet}^A_{i-1}, addr_{set}, pkey^A_{set}, pkey^B_{set}) = \Gamma^A(\mathsf{id})$
2. Extract $v_{A,i}$ and $v_{B,i}$ from $\overrightarrow{\theta}$, and c from tx_F
3. Let $\mathsf{tx}_{set_i} \leftarrow \mathtt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$

4. Let $\overrightarrow{\mathsf{tid}} := (\mathsf{tx}^A_{\mathsf{Pay},i}.\mathsf{id},\ \mathsf{tx}^B_{\mathsf{Pay},i}.\mathsf{id})$ be a tuple of the transaction ids of transaction $\mathsf{tx}^A_{\mathsf{Pay},i}$ and $\mathsf{tx}^B_{\mathsf{Pay},i}$.
5. $(\texttt{SETUP}, \mathsf{id}, \overrightarrow{\mathsf{tid}}) \xhookrightarrow{t_0+2} \mathcal{E}$
6. If $(\texttt{SETUP–OK}, \mathsf{id}) \xhookleftarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{E}$, send $(\text{updateCom}, \mathsf{id}) \xhookrightarrow{t_1} B$
7. Wait one round.
8. $\texttt{SignTxs}^A(\mathsf{tx}_{seti}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$

Party B upon $(\text{updateCom}, \mathsf{id}) \xhookleftarrow{\tau_1 \leq \tau_0+2+t_{\mathsf{stp}}} A$

9. $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{E}$
10. If not $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookleftarrow{\tau_1} \mathcal{E}$, go idle.
11. $\texttt{SignTxs}^A(\mathsf{tx}_{seti}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$

Party A in round $t_1 + 1 + t_s$

12. If sigSet^A_i is returned from $\texttt{SignTxs}^A$, $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{t_1+1+t_s} \mathcal{E}$. Else, execute $\texttt{L–ForceClose}(\mathsf{id})$ and go idle.
13. If not $(\texttt{REVOKE}, \mathsf{id}) \xhookleftarrow{t_1+1+t_s} \mathcal{E}$, go idle.
14. A together with B runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature. $\sigma^A_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^A_{\mathsf{Pnsh},i}$.Party A receives $\sigma^A_{\mathsf{Pnsh},i}$ as output after t_r. In case of failure, execute $\texttt{L–ForceClose}(\mathsf{id})$.
15. $(\text{REVOKE}, \mathsf{id}, \sigma^A_{\mathsf{Pnsh},i}) \xhookrightarrow{t_1+1+t_s+t_r} B$

Party B in round $\tau_1 + t_s$

16. If sigSet^B_i is not returned from $\texttt{SignTxs}^A$, execute $\texttt{L–ForceClose}(\mathsf{id})$ and go idle.
17. Participate in the signing of $\mathsf{tx}^A_{\mathsf{Pnsh},i}$.
18. Upon $(\text{REVOKE}, \mathsf{id}, \sigma^A_{\mathsf{Pnsh},i}) \xhookleftarrow{\tau_1+1+t_s+t_r} A$, continue. Else, execute $\texttt{L–ForceClose}(\mathsf{id})$ and go idle.
19. $(\texttt{REVOKE–REQ}, \mathsf{id}) \xhookrightarrow{\tau_1+1+t_s+t_r} \mathcal{E}$
20. If not $(\texttt{REVOKE}, \mathsf{id}) \xhookleftarrow{\tau_1+1+t_s+t_r} \mathcal{E}$, go idle.
21. B together with A runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature. $\sigma^B_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^B_{\mathsf{Pnsh},i}$. Party B receives $\sigma^B_{\mathsf{Pnsh},i}$ as output after t_r. In case of failure, execute $\texttt{L–ForceClose}(\mathsf{id})$.
22. $(\text{REVOKE}, \mathsf{id}, \sigma^B_{\mathsf{Pnsh},i}) \xhookrightarrow{\tau_1+1+t_s+2t_r} A$
23. $\Theta^B(\mathsf{id}) := \Theta^B \cup \{(\mathsf{tx}_{seti-1}, \mathsf{sigSet}^B_{i-1}, \sigma^B_{\mathsf{Pnsh},i-1})\}$
24. $\Gamma^B(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{seti}, \mathsf{sigSet}^B_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$
25. $(\texttt{UPDATED}, \mathsf{id}) \xhookrightarrow{\tau_1+2+t_s+2t_r} \mathcal{E}$

Party A in round $t_1 + 2 + t_s + t_r$

26. Participate in the signing of $\mathsf{tx}^{B}_{\mathsf{Pnsh},i}$.
27. If $(\texttt{REVOKE}, \mathsf{id}, \sigma^{B}_{\mathsf{Pnsh},i}) \xleftarrow{t_1+3+t_s+2t_r} B$ and the signature is valid, go to next step. Else, execute $\texttt{L–ForceClose}(\mathsf{id})$.
28. $\Theta^A(\mathsf{id}) := \Theta^A \cup \{(\mathsf{tx}_{set\,i-1}, \mathsf{sigSet}^A_{i-1}, \sigma^B_{\mathsf{Pnsh},i-1})\}$
29. $\Gamma^A(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{set\,i}, \mathsf{sigSet}^A_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$
30. $(\texttt{UPDATED}, \mathsf{id}) \xhookrightarrow{t_1+3+t_s+2t_r} \mathcal{E}$

Close

Party A upon $(\texttt{CLOSE}, \mathsf{id}) \xhookleftarrow{t_0} \mathcal{E}$

1. Extract $(\mathsf{tx}_F, \mathsf{tx}_{set\,i}, \mathsf{sigSet}^A_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$ from $\Gamma^A(\mathsf{id})$.
2. Extract $v_{A,i}$ and $v_{B,i}$ from $\mathsf{tx}^A_{\mathsf{Pay},j} \in \mathsf{tx}_{set\,i}$, and c from tx_F
3. Create transaction $\mathsf{tx}_c := \mathsf{tx}(\mathsf{Ch}_{AB}, \{\mathsf{pk}_A, \mathsf{pk}_B\}, \{v_{A,i}+c, v_{B,i}+c\})$, where pk_A is an address controlled by A and pk_B an address controlled by B.
4. A together with B runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature, σ_{tx_c} on the transaction tx_c. This takes t_r rounds.
5. In case the signature generation was successful, post $(\mathsf{tx}_c, \sigma_{\mathsf{tx}_c})$ on $\mathcal{L}$. Else, execute $\texttt{L–ForceClose}(\mathsf{id})$.
6. If tx_c appears on $\mathcal{L}$ in round $t_1 \leq t_0 + t_r + \Delta$, set $\Theta^A(\mathsf{id}) := \bot$, $\Gamma^A(\mathsf{id}) := \bot$ and send $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{t_2} \mathcal{E}$.

Punish

Party A upon $\texttt{PUNISH} \xhookleftarrow{t_0} \mathcal{E}$:

For each $\mathsf{id} \in \{0,1\}^*$ s.t. $\Theta^P(\mathsf{id}) \neq \bot$:

1. Iterate over all elements $(\mathsf{tx}_{set\,i}, \mathsf{sigSet}^A_i, \sigma^B_{\mathsf{Pnsh},i})$ in $\Theta^P(\mathsf{id})$
2. If the revoked payment $\mathsf{tx}^B_{\mathsf{Pay},i} \in \mathsf{tx}_{set\,i}$ is on $\mathcal{L}$, post $\left(\mathsf{tx}^B_{\mathsf{Pnsh},i}, \sigma^B_{\mathsf{Pnsh},i}\right)$ on $\mathcal{L}$ before the absolute timeout $\mathbf{T}$.
3. Let $\mathsf{tx}^B_{\mathsf{Pnsh},i}$ be accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + \Delta$. Post $(\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}, \sigma_{\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}} \in \mathsf{sigSet}^A_i)$
4. After $\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$, set $\Theta^A(\mathsf{id}) := \bot$, $\Gamma^A(\mathsf{id}) := \bot$ and output $(\texttt{PUNISHED}, \mathsf{id}) \xhookrightarrow{t_1} \mathcal{E}$.

Subprotocols

`L–ForceClose`(id):

Let t_0 be the current round

1. Extract $(\mathsf{tx}_F, \mathsf{tx}_{set0}, \mathsf{sigSet}_0^A, addr_{set}, pkey_{set}^A, pkey_{set}^B)$ from $\Gamma^A(\mathsf{id})$ and extract $\mathsf{tx}_{\mathsf{Pay},j}^A$ from tx_{set} and $\sigma_{\mathsf{Pay},j}^A$ and sigSet.
2. Party A posts $\left(\mathsf{tx}_{\mathsf{Pay},j}^A, \sigma_{\mathsf{Pay},j}^A\right)$ on $\mathcal{L}$
3. Let $t_1 \leq t_0 + \Delta$ be the round in which $\mathsf{tx}_{\mathsf{Pay},j}^A$ is accepted by $\mathcal{L}$.
4. If $\mathsf{tx}_{\mathsf{Fpay},i}^{A,B}$ appears on $\mathcal{L}$ at or after round $t_2 \leq t_1 + \Delta$ and before $\mathbf{T}$, post $\left(\mathsf{tx}_{\mathsf{Pay},j}^A, \sigma_{\mathsf{Pay},j}^A\right)$ and send $(\texttt{CLOSED}, \mathsf{id}) \xrightarrow{t_3 \leq t_2 + \Delta} \mathcal{E}$. Otherwise, post $\left(\mathsf{tx}_{\mathsf{Fpay},i}^{A,A}, \sigma_{\mathsf{Fpay},i}^{A,A}\right)$ after $\mathbf{T}$ and send $(\texttt{CLOSED}, \mathsf{id}) \xrightarrow{t_4 \leq \mathbf{T} + \Delta} \mathcal{E}$.
5. Set $\Gamma^P(\mathsf{id}) := \bot$, $\Theta^P(\mathsf{id}) := \bot$.

`GenerateTxs`$(addr_{set}, pkey_{set}^A, pkey_{set}^B, c, v_{A_i}, v_{B_i})$:

1. Using the addresses in $addr_{set}$ and the public keys in $pkey_{set}^A$ and $pkey_{set}^B$, do the following.
2. Generate $\mathsf{tx}_{\mathsf{Pay},i}^A := \mathsf{tx}(\mathsf{Ch}_{AB}, [\mathsf{pk}_{\mathsf{CPay},A}, \mathsf{SleepyCh}_A, \mathsf{ExitCh}_B], [c, v_{A,i}, v_{B,i} + c])$
3. Generate $\mathsf{tx}_{\mathsf{Pay},i}^B := \mathsf{tx}(\mathsf{Ch}_{AB}, [\mathsf{pk}_{\mathsf{CPay},B}, \mathsf{SleepyCh}_B, \mathsf{ExitCh}_A] [c, v_{B,i}, v_{A,i} + c])$
4. Generate punishment transactions $\mathsf{tx}_{\mathsf{Pnsh},i}^A := \mathsf{tx}(\mathsf{SleepyCh}_A, \mathsf{pk}_{\mathsf{pun},B}, v_{A,i})$ and $\mathsf{tx}_{\mathsf{Pnsh},i}^B := \mathsf{tx}(\mathsf{SleepyCh}_B, \mathsf{pk}_{\mathsf{pun},A}, v_{B,i})$
5. Generate finish-pay transactions $\mathsf{tx}_{\mathsf{Fpay},i}^{A,A} := \mathsf{tx}(\mathsf{SleepyCh}_A, \mathsf{pk}_{\mathsf{fp},A}, v_{A,i})$ and $\mathsf{tx}_{\mathsf{Fpay},i}^{B,B} := \mathsf{tx}(\mathsf{SleepyCh}_B, \mathsf{pk}_{\mathsf{fp},B}, v_{B,i})$ both timelocked until time $\mathbf{T}$.
6. Generate a set of faster finish-pay transactions $\mathsf{tx}_{\mathsf{Fpay},i}^{A,B} := \mathsf{tx}(\mathsf{ExitCh}_A, [\mathsf{pk}_{\mathsf{ffp},B}, \mathsf{aux}_A], [v_{B,i} + c - \epsilon, \epsilon])$ and $\mathsf{tx}_{\mathsf{Fpay},i}^{B,A} := \mathsf{tx}(\mathsf{ExitCh}_B, [\mathsf{pk}_{\mathsf{ffp},A}, \mathsf{aux}_B], [v_{A,i} + c - \epsilon, \epsilon])$.
7. Generate a set of enabler transactions $\mathsf{tx}_{\mathsf{Fpay},i}^{A*} := \mathsf{tx}([\mathsf{SleepyCh}_A, \mathsf{aux}_A], \mathsf{pk}_{\mathsf{fp},A}, v_{A,i} + \epsilon)$ and $\mathsf{tx}_{\mathsf{Fpay},i}^{B*} := \mathsf{tx}([\mathsf{SleepyCh}_B, \mathsf{aux}_B], \mathsf{pk}_{\mathsf{fp},B}, v_{B,i} + \epsilon)$ that enable a faster finish-payment.
8. Return $\mathsf{tx}_{set} := \{\mathsf{tx}_{\mathsf{Pay},i}^A, \mathsf{tx}_{\mathsf{Pay},i}^B, \mathsf{tx}_{\mathsf{Pay},i}^A, \mathsf{tx}_{\mathsf{Pnsh},i}^A, \mathsf{tx}_{\mathsf{Pnsh},i}^B, \mathsf{tx}_{\mathsf{Fpay},i}^{A,A}, \mathsf{tx}_{\mathsf{Fpay},i}^{B,B}, \mathsf{tx}_{\mathsf{Fpay},i}^{A,B}, \mathsf{tx}_{\mathsf{Fpay},i}^{B,A}, \mathsf{tx}_{\mathsf{Fpay},i}^{A*}, \mathsf{tx}_{\mathsf{Fpay},i}^{B*}\}$

$\mathtt{SignTxs}^A(\mathsf{tx}_{set}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$:
Party A (specified by the superscript of the function) is the one that receives the signatures first.
Upon agreement, i.e., A and B start executing this subprotocol in the same round with the same parameters, the following is executed. Extracting the transactions, addresses and public keys from the parameters, Party A together with B runs $\mathcal{F}_{\mathsf{Sign}}$ to sign the transactions as follows.

1. Party A receives signature $\sigma^{A,A}_{\mathsf{Fpay},i}$ on transaction $\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}$ under the shared key $\mathsf{SleepyCh}_A$.
2. Party B receives signature $\sigma^{B,B}_{\mathsf{Fpay},i}$ on transaction $\mathsf{tx}^{B,B}_{\mathsf{Fpay},i}$ under the shared key $\mathsf{SleepyCh}_B$.
3. Party A receives signatures $\left(\sigma_{\mathsf{SleepyCh},A}, \sigma_{\mathsf{aux},A}\right)$ on the transaction $\mathsf{tx}^{A*}_{\mathsf{Fpay},i}$ with respect to the shared keys $\mathsf{SleepyCh}_A$ and aux_A, respectively.
4. Party B receives signatures $\left(\sigma_{\mathsf{SleepyCh},B}, \sigma_{\mathsf{aux},B}\right)$ on the transaction $\mathsf{tx}^{B*}_{\mathsf{Fpay},i}$ with respect to the shared keys $\mathsf{SleepyCh}_B$ and aux_B, respectively.
5. Party A receives signature $\sigma^{A,B}_{\mathsf{Fpay},i}$ on the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ under the shared key ExitCh_B.
6. Party B receives signature $\sigma^{B,A}_{\mathsf{Fpay},i}$ on the transaction $\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}$ under the shared key ExitCh_A.
7. Party A receives signature $\sigma^{A}_{\mathsf{Pay},i}$ on the transaction $\mathsf{tx}^{A}_{\mathsf{Pay},i}$ under the shared key Ch_{AB}.
8. Party B receives signature $\sigma^{B}_{\mathsf{Pay},i}$ on the transaction $\mathsf{tx}^{B}_{\mathsf{Pay},i}$ under the shared key Ch_{AB}.

This takes t_s rounds and in case of failure (i.e., a signatures is not received or not valid for the specified transaction and output), execute the steps in Close. In case of success, returns to A $\mathsf{sigSet}^A_i := \{\sigma^{A,A}_{\mathsf{Fpay},i}, \left(\sigma_{\mathsf{SleepyCh},A}, \sigma_{\mathsf{aux},A}\right), \sigma^{B,A}_{\mathsf{Fpay},i}, \sigma^{A}_{\mathsf{Pay},i}\}$ and to B $\mathsf{sigSet}^B_i := \{\sigma^{B,B}_{\mathsf{Fpay},i}, \left(\sigma_{\mathsf{SleepyCh},B}, \sigma_{\mathsf{aux},B}\right), \sigma^{A,B}_{\mathsf{Fpay},i}, \sigma^{B}_{\mathsf{Pay},i}\}$

Indistinguishability. What is left at this point is to show that the UC version of the protocol is computationally indistinguishable from the one described in Section 2.5. More specifically, in the UC version of the protocol we substituted (i) the 2-party key generation protocol Γ_{JKGen} for a signature scheme Π_{DS} with an idealized version $\mathcal{F}_{\mathsf{JKGen}}$ and (ii) the 2-party signing protocol Γ_{Sign} for a signature scheme Π_{DS} with an idealized version $\mathcal{F}_{\mathsf{Sign}}$. For the UC formulations we refer the reader to [BH04, Can00]. Let Π'' be the protocol we presented in Section 2.5.

Π'. We define Π' as Π'' except that the (UC-secure) 2-party key generation protocol Γ_{JKGen} for a signature scheme Π_{DS} is replaced by an idealized version $\mathcal{F}_{\mathsf{JKGen}}$. Such ideal functionality samples a key pair honestly and simulates the shares of the corrupted party.

$\Pi'' \approx \Pi'$. Towards a contradiction, we assume that there exists an adversary $\mathcal{A}$ that can computationally distinguish between Π' and Π''. We can construct a reduction algorithm $\mathcal{R}$ that uses $\mathcal{A}$ as a subprocedure. Since the two protocols only differ in Γ_{JKGen} being replaced by $\mathcal{F}_{\mathsf{JKGen}}$, $\mathcal{R}$ using $\mathcal{A}$ can be used to distinguish a keyshare of Γ_{JKGen} from the data received in $\mathcal{F}_{\mathsf{JKGen}}$, which in turn would break the security of our 2-party key generation protocol with non-negligible probability.

Π. We define Π as Π' except that the (UC-secure) 2-party signing protocol Γ_{Sign} for a signature scheme Π_{DS} is replaced with an idealized version $\mathcal{F}_{\mathsf{Sign}}$, which signs messages locally and simulates the interaction of corrupted parties. Note that this corresponds to the UC version of the protocol.

$\Pi' \approx \Pi$. Towards a contradiction, we assume that there exists an adversary $\mathcal{A}$ that can computationally distinguish between Π and Π'. Since the two protocols only differ in Γ_{Sign} being replaced by $\mathcal{F}_{\mathsf{Sign}}$, this means that $\mathcal{A}$ is able to distinguish a real interaction from a simulated one with non-negligible probability. This is a contradiction against the UC-security of Γ_{Sign}.

A.1.1 UC Simulator

In this section, we give the pseudocode of a simulator for the formal Sleepy Channel protocol Π of Appendix A.1 in the ideal world. Our simulator interacts with $\mathcal{F}_L$ and $\mathcal{L}$. The subprotocol $\texttt{SignTxs}^P$ refers to the one given in the formal protocol description. Normally, the challenge of providing a UC-simulation proof is that the simulator is not given the secret inputs of parties sent by the environment. Instead, the functionality usually specifies exactly what is leaked to the simulator, and the simulator has to generate a simulated transcript merely from this leaked information. The simulated transcript has to be indistinguishable from the transcript that is the result of the real-world protocol execution.

Note that in our model, all messages to the functionality are implicitly forwarded to the simulator, i.e., there are no secret inputs. Hence, we can omit the simulation of the case where both protocol participants are honest; the simulator in this case would merely need to recreate the side-effect of the protocol code, which can be easily achieved with access to all the messages sent to the functionality. Indeed, the main challenge in our setting is to handle any behavior of malicious parties.

Simulator for Create

Case A is honest and B is corrupted

Upon A sending $(\texttt{CREATE}, \gamma, \mathsf{tid}_A) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$, if B does not send $(\texttt{CREATE}, \gamma, \mathsf{tid}_B) \overset{\tau}{\hookrightarrow} \mathcal{F}_L$ where $|\tau_0 - \tau| \leq T_1$, then distinguish the following cases:

1. If B sends $(\text{createInfo}, \mathsf{id}, \mathsf{tid}_B, pkey^B_{set}) \overset{\tau_0}{\hookrightarrow} A$, then send $(\texttt{CREATE}, \gamma, \mathsf{tid}_B) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ on behalf of B.
2. Otherwise stop.

Do the following:

1. Set $\mathsf{id} := \gamma.\mathsf{id}$, generate $\left(\mathsf{pk}_{\mathsf{CPay},A}, \mathsf{sk}_{\mathsf{CPay},A}\right)$, $(\mathsf{pk}_{\mathsf{pun},A}, \mathsf{sk}_{\mathsf{pun},A})$, $(\mathsf{pk}_{\mathsf{fp},A}, \mathsf{sk}_{\mathsf{fp},A})$ and $(\mathsf{pk}_{\mathsf{ffp},A}, \mathsf{sk}_{\mathsf{ffp},A})$. Let $pkey^A_{set}$ be the set of public keys of these key pairs. Send $(\text{createInfo}, \mathsf{id}, \mathsf{tid}_A,$

$pkey^A_{set}) \xhookrightarrow{\tau_0} B$.

2. If you receive $(\text{createInfo}, \mathsf{id}, \mathsf{tid}_B, pkey^B_{set}) \xhookleftarrow{\tau_0+1} B$, do the following. Else go idle.
3. Using $pkey^A_{set}$ and $pkey^B_{set}$, the simulator on behalf of A together with B runs $\mathcal{F}_{\mathsf{JKGen}}$ to generate the following set of shared addresses: $addr_{set} := \{\mathsf{Ch}_{AB}, \mathsf{SleepyCh}_A, \mathsf{SleepyCh}_B, \mathsf{ExitCh}_A, \mathsf{ExitCh}_B, \mathsf{aux}_A, \mathsf{aux}_B\}$ which takes t_g rounds. In case of failure, abort.
4. Generate $\mathsf{tx}_f := \mathsf{tx}([\mathsf{tid}_A, \mathsf{tid}_B], [\mathsf{Ch}_{AB}], [2 \cdot c + v_{A,0} + v_{B,0}])$
5. Let $\mathsf{tx}_{set0} \leftarrow \texttt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$
6. Let $\mathsf{sigSet}^A_0 \leftarrow \texttt{SignTxs}^A(\mathsf{tx}_{set0}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$
7. Generates a signature on behalf of A, σ_{tid_A}, for the output tid_A and send $(\text{createFund}, \mathsf{id}, \sigma_{\mathsf{tid}_A}) \xhookrightarrow{t_0+1+t_g+t_s} A$.
8. If you $(\text{createFund}, \mathsf{id}, \sigma_{\mathsf{tid}_B}) \xhookleftarrow{\tau_0+2+t_g+t_s} B$, post $(\mathsf{tx}_F, \{\sigma_{\mathsf{tid}_A}, \sigma_{\mathsf{tid}_B}\})$ to $\mathcal{L}$.
9. If tx_F is accepted by $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + 2 + t_g + t_s + \Delta$, store $\Gamma^A(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{set0}, \mathsf{sigSet}^A_0, addr_{set}, pkey^A_{set}, pkey^B_{set})$.

Simulator for Update

Case A is honest and B is corrupted

Upon A sending $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0} \mathcal{F}_L$, proceed as follows:

1. $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{t_0} B$
2. Upon $(\text{updateInfo}, \mathsf{id}) \xhookleftarrow{t_0+2} B$, do the following
3. Retrieve $(\mathsf{tx}_F, \mathsf{tx}_{set_{i-1}}, \mathsf{sigSet}^A_{i-1}, addr_{set}, pkey^A_{set}, pkey^B_{set}) = \Gamma^A(\mathsf{id})$
4. Extract $v_{A,i}$ and $v_{B,i}$ from $\overrightarrow{\theta}$, and c from tx_F
5. Let $\mathsf{tx}_{set_i} \leftarrow \texttt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$
6. Let $\overrightarrow{\mathsf{tid}} := (\mathsf{tx}^A_{\mathsf{Pay},i}.\mathsf{id}, \mathsf{tx}^B_{\mathsf{Pay},i}.\mathsf{id})$ be a tuple of the transaction ids of transaction $\mathsf{tx}^A_{\mathsf{Pay},i}$ and $\mathsf{tx}^B_{\mathsf{Pay},i}$. Inform $\mathcal{F}_L$ of $\overrightarrow{\mathsf{tid}}$ in round $t_0 + 2$.
7. If A sends $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{t_1 \leq t_0+2+t_{\mathsf{stp}}} \mathcal{F}_L$, send $(\text{updateCom}, \mathsf{id}) \xhookrightarrow{t_1} B$
8. Wait one round.
9. If in round $t_1 + 1$, B starts executing $\texttt{SignTxs}^A(\mathsf{tx}_{set_i}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$, send $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{t_1+1} \mathcal{F}_L$ on behalf of B
10. $\texttt{SignTxs}^A(\mathsf{tx}_{set_i}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$
11. If sigSet^A_i is returned from $\texttt{SignTxs}^A$, instruct $\mathcal{F}_L$ to $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{t_1+1+t_s} \mathcal{E}$ via A. Else, execute $\texttt{L–ForceClose}^A(\mathsf{id})$ and go idle.

12. If A does not send $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{t_1+1+t_s} \mathcal{F}_L$, go idle.
13. The simulator on behalf of A together with B runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature. $\sigma^A_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^A_{\mathsf{Pnsh},i}$. Party A receives $\sigma^A_{\mathsf{Pnsh},i}$ as output. This takes t_r rounds. In case of failure, execute $\texttt{L–ForceClose}^A(\mathsf{id})$.
14. $(\texttt{REVOKE}, \mathsf{id}, \sigma^A_{\mathsf{Pnsh},i}) \xhookrightarrow{t_1+1+t_s+t_r} B$
15. If B starts $\mathcal{F}_{\mathsf{Sign}}$ to sign $\mathsf{tx}^B_{\mathsf{Pnsh},i}$ in round $t_1 + 2 + t_s + t_r$, send $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{t_1+2+t_s+t_r} \mathcal{F}_L$ on behalf of B and participate in the signing on behalf of A.
16. If $(\texttt{REVOKE}, \mathsf{id}, \sigma^B_{\mathsf{Pnsh},i}) \xhookleftarrow{t_1+3+t_s+2t_r} B$ and the signature is valid, go to next step. Else, execute $\texttt{L–ForceClose}^A(\mathsf{id})$.
17. $\Theta^A(\mathsf{id}) := \Theta^A \cup \{(\mathsf{tx}_{set_{i-1}}, \mathsf{sigSet}^A_{i-1}, \sigma^B_{\mathsf{Pnsh},i-1})\}$
18. $\Gamma^A(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{set_i}, \mathsf{sigSet}^A_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$

Case B is honest and A is corrupted

Upon A sending $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{t_0} B$, send $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{t_0} \mathcal{F}_L$ on behalf of A, if A has not already sent this message. Proceed as follows:

1. Upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookleftarrow{\tau_0} A$, do the following
2. Retrieve $(\mathsf{tx}_F, \mathsf{tx}_{set_{i-1}}, \mathsf{sigSet}^B_{i-1}, addr_{set}, pkey^A_{set}, pkey^B_{set}) = \Gamma^B(\mathsf{id})$
3. Extract $v_{A,i}$ and $v_{B,i}$ from $\overrightarrow{\theta}$, and c from tx_F
4. Let $\mathsf{tx}_{set_i} \leftarrow \texttt{GenerateTxs}(addr_{set}, pkey^A_{set}, pkey^B_{set}, c, v_{A_i}, v_{B_i})$
5. Let $\overrightarrow{\mathsf{tid}} := (\mathsf{tx}^A_{\mathsf{Pay},i}.\mathsf{id}, \mathsf{tx}^B_{\mathsf{Pay},i}.\mathsf{id})$ be a tuple of the transaction ids of transaction $\mathsf{tx}^A_{\mathsf{Pay},i}$ and $\mathsf{tx}^B_{\mathsf{Pay},i}$. Inform $\mathcal{F}_L$ of $\overrightarrow{\mathsf{tid}}$.
6. $(\text{updateInfo}, \mathsf{id}) \xhookrightarrow{\tau_0} A$
7. Upon A sending $(\text{updateCom}, \mathsf{id}) \xhookrightarrow{\tau_0+1+t_{\mathsf{stp}}} B$, send $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{F}_L$ on behalf of A.
8. Receive $(\text{updateCom}, \mathsf{id}) \xhookleftarrow{\tau_1 \leq \tau_0+2+t_{\mathsf{stp}}} A$
9. If B sends $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{F}_L$, $\texttt{SignTxs}^A(\mathsf{tx}_{set_i}, addr_{set}, pkey^A_{set} \cup pkey^B_{set})$
10. If sigSet^B_i is not returned from $\texttt{SignTxs}^A$ in round $\tau_1 + t_s$, execute $\texttt{L–ForceClose}^B(\mathsf{id})$ and go idle.
11. If A starts the $\mathcal{F}_{\mathsf{Sign}}$ in round $\tau_1 + t_s$ to generate $\sigma^A_{\mathsf{Pnsh},i}$, send $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1+t_s} \mathcal{F}_L$ on behalf of A. Participate in the signing on behalf of B.

12. Upon $(\text{REVOKE}, \mathsf{id}, \sigma^A_{\mathsf{Pnsh},i}) \xleftarrow{\tau_1+1+t_s+t_r} A$, continue. Else, execute $\texttt{L–ForceClose}^B(\mathsf{id})$ and go idle.
13. If B does not send $(\texttt{REVOKE}, \mathsf{id}) \xrightarrow{\tau_1+1+t_s+t_r} \mathcal{F}_L$, go idle.
14. $\mathcal{S}$ on behalf of B together with A runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature. $\sigma^B_{\mathsf{Pnsh},i}$ on the punishment transaction $\mathsf{tx}^B_{\mathsf{Pnsh},i}$. Party B receives $\sigma^B_{\mathsf{Pnsh},i}$ as output after t_r. In case of failure, execute $\texttt{L–ForceClose}^B(\mathsf{id})$.
15. $(\text{REVOKE}, \mathsf{id}, \sigma^B_{\mathsf{Pnsh},i}) \xrightarrow{\tau_1+1+t_s+2t_r} A$
16. $\Theta^B(\mathsf{id}) := \Theta^B \cup \{(\mathsf{tx}_{set i-1}, \mathsf{sigSet}^B_{i-1}, \sigma^B_{\mathsf{Pnsh},i-1})\}$
17. $\Gamma^B(\mathsf{id}) := (\mathsf{tx}_F, \mathsf{tx}_{set i}, \mathsf{sigSet}^B_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$

Simulator for Close

Case A is honest and B is corrupted

Upon A sending $(\texttt{CLOSE}, \mathsf{id}) \xhookrightarrow{t_0} \mathcal{F}_L$, do the following.

1. Extract $(\mathsf{tx}_F, \mathsf{tx}_{set i}, \mathsf{sigSet}^A_i, addr_{set}, pkey^A_{set}, pkey^B_{set})$ from $\Gamma^A(\mathsf{id})$.
2. Extract $v_{A,i}$ and $v_{B,i}$ from $\mathsf{tx}^A_{\mathsf{Pay},j} \in \mathsf{tx}_{set i}$, and c from tx_F
3. Create transaction $\mathsf{tx}_c := \mathsf{tx}(\mathsf{Ch}_{AB}, \{\mathsf{pk}_A, \mathsf{pk}_B\}, \{v_{A,i}+c, v_{B,i}+c\})$, where pk_A is an address controlled by A and pk_B an address controlled by B.
4. The simulator on behalf of A together with B runs the interactive protocol $\mathcal{F}_{\mathsf{Sign}}$ to generate the following signature, σ_{tx_c} on the transaction tx_c. This takes t_r rounds.
5. In case the signature generation was successful, post $(\mathsf{tx}_c, \sigma_{\mathsf{tx}_c})$ on $\mathcal{L}$ and send $(\texttt{CLOSE}, \mathsf{id}) \xhookrightarrow{t_0+t_r} \mathcal{F}_L$ on behalf of B. Else, execute $\texttt{L–ForceClose}^A(\mathsf{id})$.
6. If tx_c appears on $\mathcal{L}$ in round $t_1 \leq t_0 + t_r + \Delta$, set $\Theta^A(\mathsf{id}) := \bot$, $\Gamma^A(\mathsf{id}) := \bot$.

Simulator for Punish

Case A is honest and B is corrupted

Upon A sending $\texttt{PUNISH} \xhookrightarrow{\tau_0} \mathcal{F}_L$, for each $\mathsf{id} \in \{0,1\}^*$ such that $\Theta^A(\mathsf{id}) \neq \bot$ do the following:

1. Parse $\{(\mathsf{tx}_{set i}, \mathsf{sigSet}^A_i, \sigma^B_{\mathsf{Pnsh},i})\}_{i \in m} := \Theta^A(\mathsf{id})$ and extract γ from $\Gamma^A(\mathsf{id})$. If for some $i \in m$, there exist a transaction $\mathsf{tx}^B_{\mathsf{Pay},i} \in \mathsf{tx}_{set i}$ on $\mathcal{L}$ do the following.
2. Post $\left(\mathsf{tx}^B_{\mathsf{Pnsh},i}, \sigma^B_{\mathsf{Pnsh},i}\right)$ on $\mathcal{L}$ before the absolute timeout $\mathbf{T}$.
3. Let $\mathsf{tx}^B_{\mathsf{Pnsh},i}$ be accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + \Delta$. Post $(\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}, \sigma_{\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}} \in \mathsf{sigSet}^A_i)$

4. After $\mathsf{tx}^{B,A}_{\mathsf{Fpay},i}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$, set $\Theta^A(\mathsf{id}) := \bot$, $\Gamma^A(\mathsf{id}) := \bot$.

Simulator for $\mathtt{ForceClose}^P(\mathsf{id})$

Let τ_0 be the current round

1. Extract $(\mathsf{tx}_F, \mathsf{tx}_{set0}, \mathsf{sigSet}^A_0, addr_{set}, pkey^A_{set}, pkey^B_{set})$ from $\Gamma^A(\mathsf{id})$ and extract $\mathsf{tx}^A_{\mathsf{Pay},j}$ from tx_{set} and $\sigma^A_{\mathsf{Pay},j}$ and sigSet.
2. Post $\left(\mathsf{tx}^A_{\mathsf{Pay},j}, \sigma^A_{\mathsf{Pay},j}\right)$ on $\mathcal{L}$
3. Let $t_2 \leq t_1 + \Delta$ be the round in which $\mathsf{tx}^A_{\mathsf{Pay},j}$ is accepted by $\mathcal{L}$.
4. If $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ appears on $\mathcal{L}$ at or after round $t_3 \leq t_2 + \Delta$ and before $\mathbf{T}$, post $\left(\mathsf{tx}^A_{\mathsf{Pay},j}, \sigma^A_{\mathsf{Pay},j}\right)$. Otherwise, post $\left(\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}, \sigma^{A,A}_{\mathsf{Fpay},i}\right)$ after $\mathbf{T}$. Set $\Gamma^P(\mathsf{id}) := \bot$, $\Theta^P(\mathsf{id}) := \bot$.

A.1.2 Simulation proof

To prove that the protocol is a (G)UC-realization of the functionality $\mathcal{F}_L$, we show that the execution ensembles $EXEC_{\Pi,\mathcal{A},\mathcal{E}}$ and $EXEC_{\mathcal{F},\mathcal{S},\mathcal{E}}$ are computationally indistinguishable. I.e., for the simulator $\mathcal{S}$ presented in Appendix A.1.1, for every environment the interaction with $\mathcal{S}$ and $\mathcal{F}_L$ is computationally indistinguishable from the interaction with $\mathcal{A}$ and Π. We show this for the different phases Create, Update, Close, Punish as well as the subprotocol ForceClose.

For readability, we define $m[\tau]$ to capture the fact that a message m is observed by the environment in round τ. Note that messages sent to parties in the protocol that are under adversarial control observe the message after one round. Additionally, we interact with other functionalities, e.g., for signing and the ledger. To capture any side effect observable by the environment including messages sent by parties who are potentially controlled by the adversary or changing public variables such as the ledger, we do the following. We denote $\mathsf{obsSet}(\mathsf{action}, \tau)$ as the set of all observable side effects triggered by action action in round τ. Finally, we refer to a message by the message identifier, e.g., `CREATE` or createInfo. We note that other message parameters are omitted. Instead, we refer to relevant parts in the ideal world and the real world, where one can verify that indeed the same objects are created, checks are performed, etc.

We require a SUF-CMA secure signature scheme Σ and a ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$ where $\mathcal{V}$ allows for transaction authorization under Σ and absolute time-locks.[1] The former property is needed to ensure that the environment and malicious party cannot generate signatures on behalf of honest parties with non-negligible probability. Instead, only the simulator

[1] The necessity for time-locks can be dropped when using verifiable timed signatures (VTS) as discussed in Section 2.5.2, although we do not provide a formal analysis for such variant here.

can generate signatures on behalf of honest parties. Further, we require a ledger that supports transaction authorization under Σ and absolute time-locks for encoding our construction.

Lemma 2. *The Create phase of* Π *UC-realizes the Create phase of* $\mathcal{F}_L$.

Proof. We consider the case where A is honest and B is corrupted. Note that the reverse case is symmetric.

Real World. After receiving `CREATE` in round t_0, A sends message createInfo to B in t_0. If A receives also createInfo in $t_0 + 1$, A will perform first the action $a_0 :=$ "run address generation" in round $t_0 + 1$ and on success, create the transactions for the channel followed by $a_1 :=$ "create signatures" in round $t_0 + 1 + t_g$. If this is successful, A generates the signature for the funding tx tx_F and sends the signature via createFund to B in $t_0 + 1 + t_g + t_s$. If A receives also createFund from B in round $t_0 + 2 + t_g + t_s$, it will perform action $a_2 :=$ "Post funding tx on $\mathcal{L}$". If it is accepted in round $t_1 \leq t_0 + 2 + t_g + t_s + \Delta$, finally A will output `CREATED`. Thus, the execution ensemble is $EXEC^{\text{create}}_{\Pi,\mathcal{A},\mathcal{E}} := \{\text{createFund}[t_0{+}1], \mathsf{obsSet}(a_0, t_0{+}1), \mathsf{obsSet}(a_1, t_0{+}1{+}t_g), \text{createFund}[t_0{+}2 + t_g + t_s], \mathsf{obsSet}(a_2, t_0 + 2 + t_g + t_s), \texttt{CREATED}[t_1]\}$.

Ideal World. After A sending `CREATE` in round t_0 to $\mathcal{F}_L$, the simulator sends createInfo to B. If B sends createInfo to A, the simulator informs $\mathcal{F}_L$ and performs a_0 in round $t_0 + 1$. Upon success, $\mathcal{S}$ creates the transactions for the channel and performs a_1 in round $t_0 + 1 + t_g$. If this was successful, the simulator on behalf of A generates the signature of tx_F and sends createFund to B in $t_0 + 1 + t_g + t_s$. If B sends also createFund to A, received in $t_0 + 2 + t_g + t_s + \Delta$, perform a_2 in $t_0 + 2 + t_g + t_s + \Delta$. If the funding tx is accepted in round $t_1 \leq t_0 + 2 + t_g + t_s + \Delta$, $\mathcal{F}_L$ (which expects it after being informed by $\mathcal{S}$) outputs `CREATED` in round $t_1 \leq t_0 + 2 + t_g + t_s + \Delta$. Thus, the execution ensemble is $EXEC^{\text{create}}_{\mathcal{F},\mathcal{S},\mathcal{E}} := \{\text{createFund}[t_0{+}1], \mathsf{obsSet}(a_0, t_0{+}1), \mathsf{obsSet}(a_1, t_0{+}1{+}t_g), \text{createFund}[t_0{+}2 + t_g + t_s], \mathsf{obsSet}(a_2, t_0 + 2 + t_g + t_s), \texttt{CREATED}[t_1]\}$ □

Lemma 3. *The ForceClose subprotocol of* Π *UC-realizes the ForceClose subprocedure of* $\mathcal{F}_L$.

Proof. We consider the case where A is honest and B is corrupted. Note that the reverse case is symmetric.

Real World. Taking the latest state, a performs action $a_0 :=$ "post $\left(\mathsf{tx}^A_{\mathsf{Pay},j}, \sigma^A_{\mathsf{Pay},j}\right)$ on $\mathcal{L}$" in round t_0. After the transaction appears on $\mathcal{L}$ in round $t_1 \leq t_0 + \Delta$, do the following depending on B. Either (i) the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ appears on $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$ and before $\mathbf{T}$. In this case, A posts $\left(\mathsf{tx}^A_{\mathsf{Pay},j}, \sigma^A_{\mathsf{Pay},j}\right)$, which we denote as action a_1, followed by sending `CLOSED` in round $t_m := t_3 leqt_2 + \Delta$. Otherwise, (ii) A posts $\left(\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}, \sigma^{A,A}_{\mathsf{Fpay},i}\right)$ after $\mathbf{T}$, which we denote as action a_2, followed by sending

$\texttt{CLOSED}$ in round $t_m := t_4 \leq \mathbf{T} + \Delta$. Thus, the execution ensemble is $EXEC^{\text{forceclose}}_{\Pi,\mathcal{A},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), o \in \{\mathsf{obsSet}(a_1, t_2), \mathsf{obsSet}(a_2, \mathbf{T})\}, \texttt{CLOSED}[t_m]\}$.

Ideal World. Taking the latest state, the simulator will mirror the behavior of the real world. In round t_0, it will perform action a_0. After the transaction appears on $\mathcal{L}$ in round $t_1 \leq t_0 + \Delta$, do the following depending on B. Either (i) the transaction $\mathsf{tx}^{A,B}_{\mathsf{Fpay},i}$ appears on $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$ and before $\mathbf{T}$. In this case, the simulator posts $\left(\mathsf{tx}^{A}_{\mathsf{Pay},j}, \sigma^{A}_{\mathsf{Pay},j}\right)$, which we denote as action a_1. Otherwise, (ii) the simulator posts $\left(\mathsf{tx}^{A,A}_{\mathsf{Fpay},i}, \sigma^{A,A}_{\mathsf{Fpay},i}\right)$ after $\mathbf{T}$, which we denote as action a_2.Meanwhile, the functionality $\mathcal{F}_L$ expects that either of these transactions appears on $\mathcal{L}$. If this happens, either in round $t_m := t_3 \leq t_2 + \Delta$ in case (i) or in round $t_m := t_4 \leq \mathbf{T} + \Delta$, it outputs $\texttt{CLOSED}$. Thus, the execution ensemble is $EXEC^{\text{forceclose}}_{\mathcal{F},\mathcal{S},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), o \in \{\mathsf{obsSet}(a_1, t_2), \mathsf{obsSet}(a_2, \mathbf{T})\}, \texttt{CLOSED}[t_m]\}$. □

Lemma 4. *The Update phase of* Π *UC-realizes the Update phase of* $\mathcal{F}_L$.

Proof. We start by considering the case where A is honest and B is corrupted.

Real World. A upon $\texttt{UPDATE}$ in round t_0 does the following. The update phase consists of the following steps: Informing B, generating the transactions for the new state, signing these transactions, signing the revocation for B, and signing the revocation for A. We capture the steps visible to the $\mathcal{E}$ below, together with their dependencies. The execution ensemble $EXEC^{\text{update}}_{\Pi,\mathcal{A},\mathcal{E}}$ follows as a list for better readability.

- updateReq to B in round t_0
- $\texttt{SETUP}$ to $\mathcal{E}$ in $t_0 + 2$ (if received updateInfo from B)
- updateCom to B in round $t_1 \leq t_0 + 2 + t_{\mathsf{stp}}$(if received $\texttt{SETUP–OK}$ from $\mathcal{E}$)
- $\texttt{SignTxs}$ in $t_1 + 1$
- $\texttt{UPDATE–OK}$ to $\mathcal{E}$ in round $t_1 + 1 + t_s$ (if signing successful)
- sign revocation of B with B in round $t_1 + 1 + t_s$ (if $\texttt{REVOKE}$ from $\mathcal{E}$)
- $\texttt{REVOKE}$ to B in round $t_1 + 1 + t_s + t_r$ (if signing successful)
- sign revocation of A with B in round $t_1 + 2 + t_s + t_r$
- $\texttt{UPDATED}$ to $\mathcal{E}$ in round $t_1 + 3 + t_s + 2t_r$ (if signature for revocation received from B)

Ideal World. Upon A sending $\texttt{UPDATE}$ in round t_0 to $\mathcal{F}_L$, $\mathcal{S}$ simulates the protocol view to $\mathcal{E}$. The same steps of the update phase have to be conducted: Informing B, generating the transactions for the new state, signing these transactions, signing the revocation for B, and signing the revocation for A. We capture the steps visible to the $\mathcal{E}$ below, together with their dependencies and if they are executed by $\mathcal{S}$ or $\mathcal{F}_L$. The execution ensemble $EXEC^{\text{update}}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ follows as a list for better readability.

- updateReq to B in round t_0 ($\mathcal{S}$)

- SETUP to $\mathcal{E}$ in $t_0 + 2$ (if received updateInfo from B) ($\mathcal{F}_L$)
- updateCom to B in round $t_1 \leq t_0 + 2 + t_{\mathsf{stp}}$ (if received SETUP–OK from $\mathcal{E}$) ($\mathcal{S}$)
- SignTxs in $t_1 + 1$ ($\mathcal{S}$)
- UPDATE–OK to $\mathcal{E}$ in round $t_1 + 1 + t_s$ (if signing successful) ($\mathcal{F}_L$ after instructed by $\mathcal{S}$)
- sign revocation of B with B in round $t_1 + 1 + t_s$ (if REVOKE from $\mathcal{E}$) ($\mathcal{S}$)
- REVOKE to B in round $t_1 + 1 + t_s + t_r$ (if signing successful) ($\mathcal{S}$
- sign revocation of A with B in round $t_1 + 2 + t_s + t_r$ ($\mathcal{S}$)
- UPDATED to $\mathcal{E}$ in round $t_1 + 3 + t_s + 2t_r$ (if signature for revocation received from B) ($\mathcal{F}_L$)

Table A.1: Overhead for operations, given a current fee of 102 satoshi per byte and a price of 57, 202 USD per BTC.

	txs off-chain	bytes	txs on-chain	bytes	USD
create	$2 \cdot (\mathsf{tx}^{A}_{\mathsf{Pay},i} + \mathsf{tx}^{A,B}_{\mathsf{Fpay},i} + \mathsf{tx}^{A*}_{\mathsf{Fpay},i} + \mathsf{tx}^{A,A}_{\mathsf{Fpay},i})$	2026	tx_F	338	2.13
update	$2 \cdot (\mathsf{tx}^{A}_{\mathsf{Pay},i} + \mathsf{tx}^{A,B}_{\mathsf{Fpay},i} + \mathsf{tx}^{A*}_{\mathsf{Fpay},i} + \mathsf{tx}^{A,A}_{\mathsf{Fpay},i} + \mathsf{tx}^{A}_{\mathsf{Pnsh},i})$	2408	-	-	-
close (optimistic)	-	-	$\mathsf{tx}^{A}_{\mathsf{Pay},i}$	225	1.42
close (slow)	-	-	$\mathsf{tx}^{A}_{\mathsf{Pay},i} + \mathsf{tx}^{A,A}_{\mathsf{Fpay},i}$	449	2.83
close (fast)	-	-	$\mathsf{tx}^{A}_{\mathsf{Pay},i} + \mathsf{tx}^{A,B}_{\mathsf{Fpay},i} + \mathsf{tx}^{A*}_{\mathsf{Fpay},i}$	823	5.18
punish	-	-	$\mathsf{tx}^{A}_{\mathsf{Pay},i} + \mathsf{tx}^{A}_{\mathsf{Pnsh},i}$	450	2.83

Now we consider the case where B is honest and A is corrupted.

Real World. A upon UPDATE in round t_0 does the following. The update phase consists of the following steps: Generating the transactions for the new state, signing these transactions, signing the revocation for A, and signing the revocation for B. Similar to the previous case, we capture the steps visible to the $\mathcal{E}$ below, together with their dependencies. The execution ensemble $EXEC^{\text{update}}_{\Pi,\mathcal{A},\mathcal{E}}$ follows as a list for better readability.

- UPDATE–REQ to $\mathcal{E}$ in round τ_0 (if received updateReq from A)
- updateInfo to A in round τ_0
- SETUP–OK to $\mathcal{E}$ in round $\tau_1 \leq \tau_0 + 2 + t_{\mathsf{stp}}$ (if received updateCom from A)
- SignTxs in τ_1
- sign revocation of B with A in round $\tau_1 + t_s$ (if previous signing was successful)
- REVOKE–REQ to $\mathcal{E}$ in round $\tau_1 + 1 + t_s$ (after receiving REVOKE from A in that round)
- sign revocation of A with A in round $\tau_1 + 1 + t_s + t_r$
- REVOKE to A in round $\tau_1 + 1 + t_s + 2t_r$ (in case revocation was signed successfully)
- UPDATED to $\mathcal{E}$ in round $\tau_1 + 2 + t_s + 2t_r$

Ideal World. Upon A sending UPDATE in round t_0 to $\mathcal{F}_L$, $\mathcal{S}$ simulates the protocol view to $\mathcal{E}$. The same steps of the update phase have to be conducted: Generating the transactions for the new state, signing these transactions, signing the revocation for B and signing the revocation for A. We capture the steps visible to the $\mathcal{E}$ below, together with their dependencies and if they are executed by $\mathcal{S}$ or $\mathcal{F}_L$. The execution ensemble $EXEC^{\text{update}}_{\mathcal{F},\mathcal{S},\mathcal{E}}$ follows as a list for better readability.

- UPDATE–REQ to $\mathcal{E}$ in round τ_0 (if received updateReq from A) ($\mathcal{F}_L$)
- updateInfo to A in round τ_0 ($\mathcal{S}$)
- SETUP–OK to $\mathcal{E}$ in round $\tau_1 \leq \tau_0 + 2 + t_{\mathsf{stp}}$ (if received updateCom from A) ($\mathcal{F}_L$)
- SignTxs in τ_1 ($\mathcal{S}$)
- sign revocation of B with A in round $\tau_1 + t_s$ (if previous signing was successful) ($\mathcal{S}$)
- REVOKE–REQ to $\mathcal{E}$ in round $\tau_1 + 1 + t_s$ (after receiving REVOKE from A in that round) ($\mathcal{F}_L$)
- sign revocation of A with A in round $\tau_1 + 1 + t_s + t_r$ ($\mathcal{S}$)
- REVOKE to A in round $\tau_1 + 1 + t_s + 2t_r$ (in case revocation was signed successfully) ($\mathcal{S}$)
- UPDATED to $\mathcal{E}$ in round $\tau_1 + 2 + t_s + 2t_r$ ($\mathcal{F}_L$)

□

Lemma 5. *The Close phase of* Π *UC-realizes the Close phase of* $\mathcal{F}_L$.

Proof. We consider the case where A is honest and B is corrupted. Note that the reverse case is symmetric.

Real World. After receiving CLOSE in round t_0, A creates a closing transaction tx_c from the latest state of the channel. A then performs action $a_0 :=$ create signature for tx_c with B. In case of success, A performs $a_1 :=$ post tx_c on $\mathcal{L}$ in round $t_0 + t_r$. If it appears in round $t_1 \leq t_0 + t_r + \Delta$, send CLOSED. If the signature generation was unsuccessful in round $t_2 \geq t_0$, A runs $a_2 :=$ ForceClose. Thus, the execution ensemble is either $EXEC^{\text{close}}_{\Pi,\mathcal{A},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_1, t_0 + t_r), \texttt{CLOSED}[t_1]\}$ or $EXEC^{\text{close}}_{\Pi,\mathcal{A},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_2, t_2)\}$.

Ideal World. In this case, after A receving CLOSE in round t_0, $\mathcal{S}$ handles creating the transaction and performing a_0 in round t_0 and a_1 in $t_0 + t_r$, while $\mathcal{F}_L$ sends CLOSED if the closing transaction appears on $\mathcal{L}$ in round $t_1 \leq t_0 + t_r + \Delta$. If the signature generation was unsuccessful in round $t_2 \geq t_0$, the simulator will perform a_2 and instruct $\mathcal{F}_L$ to do the same (by not sending CLOSE on behalf of B). Thus, the execution ensemble is $EXEC^{\text{close}}_{\mathcal{F},\mathcal{S},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_1, t_0 + t_r), \texttt{CLOSED}[t_1]\}$ or $EXEC^{\text{close}}_{\mathcal{F},\mathcal{S},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_2, t_2)\}$.

□

Lemma 6. *The Punish phase of* Π *UC-realizes the Punish phase of* $\mathcal{F}_L$.

Proof. We consider the case where A is honest and B is corrupted. Note that the reverse case is symmetric.

Real World. After A receives PUNISH from $\mathcal{E}$ in round t_0,[2] A checks if there is a transaction on the ledger that belongs to an old state of one of its channels. If yes, using the corresponding revocation secret, A performs action $a_0 :=$ post punishment transaction in round t_0. After it is accepted in round $t_1 \leq t_0 + \Delta$, A performs $a_1 :=$ post collateral unlock transaction. If that is accepted in round $t_2 \leq t_1 + \Delta$, A outputs message PUNISHED. Thus, the execution ensemble is $EXEC^{\text{punish}}_{\Pi,\mathcal{A},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_1, t_1), \texttt{PUNISHED}[t_2]\}$.

Ideal World. The ideal functionality checks at the end of every round t_0 (this is achieved by marking itself stale if not invoked by $\mathcal{E}$, see Section 2.4) if a transaction spending the funding transaction that is not the most recent state is on the ledger. If it is, and the other party is honest, it expects a punishment transaction to appear in round $t_1 \leq t_0 + \Delta$. Additionally, it expects that the collateral unlock transaction of that party appears in round $t_2 \leq t_1 + \Delta$. If both appear, $\mathcal{F}_L$ outputs PUNISHED in round t_2. Meanwhile, the simulator will take care of posting both the punishment a_0 and the collateral unlock transaction a_1 in rounds t_0 and t_1, respectively. Thus, the execution ensemble is $EXEC^{\text{punish}}_{\mathcal{F},\mathcal{S},\mathcal{E}} := \{\mathsf{obsSet}(a_0, t_0), \mathsf{obsSet}(a_1, t_1), \texttt{PUNISHED}[t_2]\}$.

□

Theorem 8. *The protocol* Π *UC-realizes the the ideal functionality* $\mathcal{F}_L$.

Proof. The proof of the theorem follows by a standard hybrid argument and an application of Lemmas 2 to 6. □

A.2 Deployment cost

To further evaluate our Sleepy Channels protocol, we want to measure the cost in terms of on-chain fees when using the protocol. Taking the numbers from Section 2.6, we do the following. To post a Bitcoin transaction to the blockchain, one has to give a certain amount of fees to the miner. This fee is dependent on the size of the transaction. At the time of writing, the fee of including a transaction to the next block is 11 satoshis per byte and the price of 1 Bitcoin in USD is 57202,30. Together with the fact that there are 10^8 satoshis in one Bitcoin, we can compute the fees in USD for each of the Sleepy Channels operations. We show our results in Table A.1.

[2] Note that we require the environment to send this message, as we defined that all security guarantees of $\mathcal{F}_L$ are lost in the case of message ERROR. However, this is exactly what happens if the environment does not give the execution token to $\mathcal{F}_L$ via PUNISH, see Section 2.4

APPENDIX B

Appendix to Chapter 3

B.1 On the Usage of the UC-Framework

To formally model the security of our construction, we use a synchronous version of the global UC framework (GUC) [CDPW07] which extends the standard UC framework [Can01] by allowing for a global setup. Since our model is essentially the same as in [DFH18, DEF+19b], parts of this section are taken verbatim from there.

Protocols and adversarial model. We consider a protocol π that runs between parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ that takes as input a security parameter 1^λ (with $\lambda \in \mathbb{N}$) and an auxiliary input $z \in \{0,1\}^*$, and who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). By corruption, we mean that $\mathcal{A}$ takes full control over P_i and learns its internal state. Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{E}$ – which represents anything "external" to the current protocol execution. The environment also observes all outputs returned by the parties of the protocol.

Modeling time and communication. We assume a synchronous communication network, which means that the execution of the protocol happens in rounds. Let us emphasize that the notion of rounds is just an abstraction that simplifies our model and allows us to argue about the time complexity of our protocols in a natural way. We follow [DEF+19b], which in turn follows [KMTZ13], and formalize the notion of rounds via an ideal functionality $\widehat{\mathcal{F}}_{clock}$ representing "the clock". At a high level, the ideal functionality requires all honest parties to indicate that they are prepared to proceed to the next round before the clock is "ticked". We treat the clock functionality as a *global* ideal functionality using the GUC model. This means that all entities are always aware of the given round.

We assume that parties of a protocol are connected via authenticated communication channels with guaranteed delivery of exactly one round. This means that if a party P sends a message m to party Q in round t, party Q receives this message in beginning of round $t+1$. In addition, Q is sure that the message was sent by party P. The adversary can see the content of the message and can reorder messages that were sent in the same round. However, it can not modify, delay or drop messages sent between parties, or insert new messages. The assumptions on the communication channels are formalized as an ideal functionality $\mathcal{F}_{GDC}$. We refer the reader to [DEF+19b] its formal description.

While the communication between two parties of a protocol takes exactly one round, all other communication – for example, between the adversary $\mathcal{A}$ and the environment $\mathcal{E}$ – takes zero rounds. For simplicity, we assume that any computation made by any entity takes zero rounds as well.

Finally, we allow our ledger channel ideal functionality to output the same message to two parties in the same round (c.f. Figure 3.4). Technically, this can be done as follows. The functionality first outputs the message to one of the parties, thereby loses its execution token. The functionality then waits for the next activation to send the message to the other party. Only once the message is sent to both parties, the ideal functionality allows the round to complete by "ticking the clock".

Handling coins. We model the money mechanics offered by UTXO cryptocurrencies, such as Bitcoin, via a *global* ideal functionality $\mathcal{L}$ using the GUC model. Our functionality is parameterized by a *delay parameter* Δ which upper bounded in the maximal number of rounds it takes to publish a valid transaction, a digital signature scheme Σ and a set $\mathcal{V}$ defining valid output conditions. We require that $\mathcal{V}$ includes signature verification w.r.t. Σ. The functionality accepts messages from a fixed set of parties $\mathcal{P}$.

The ledger functionality $\mathcal{L}$ is initiated by the environment $\mathcal{E}$ via the following steps: (1) $\mathcal{E}$ instructs the ledger functionality to generate public parameter of the signature scheme pp; (2) $\mathcal{E}$ instructs every party $P \in \mathcal{P}$ to generate a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ and submit the public key pk_P to the ledger via the message $(\text{register}, \mathsf{pk}_P)$; (3) sets the initial state of the ledger meaning that it initialize a set $\mathtt{TX}$ defining all published transactions.

Once initialized, the state of $\mathcal{L}$ is public and can be accessed by all parties of the protocol, the adversary $\mathcal{A}$ and the environment $\mathcal{E}$ via a read message. Any party $P \in \mathcal{P}$ can at any time post a transaction on the ledger via the message $(\text{post}, \mathsf{tx})$. The ledger functionality waits for at most Δ rounds (the exact number of rounds is determined by the adversary). Thereafter, the ledger verifies the validity of the transaction and adds it to the transaction set $\mathtt{TX}$. The formal description of the ledger functionality is presented in Figure B.1.

Let us emphasize that our ledger functionality is fairly simplified. In reality, parties can join and leave the blockchain system dynamically. Moreover, we completely abstract from the fact that transactions are published in blocks that are proposed by parties and the adversary. These and other features are captured by prior works, such as [BMTZ17], that provide a more accurate formalization of the Bitcoin ledger in the UC framework [Can01].

However, interaction with such ledger functionality is fairly complex. To increase the readability of our channel protocols and ideal functionality, which is the main focus of our work, we decided on this simpler ledger.

Ideal Functionality $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$

The functionality accepts messages from all parties that are in the set $\mathcal{P}$ and maintains a $\mathtt{PKI}$ for those parties. The functionality maintains the set of all accepted transactions $\mathtt{TX}$ and all unspent transaction outputs $\mathtt{UTXO}$.

Initialize public keys: Upon $(\text{register}, \mathsf{pk}_P) \overset{\tau_0}{\hookleftarrow} P$ and it is the first time P sends a registration message, add (pk_P, P) to $\mathtt{PKI}$.

Post transaction: Upon $(\text{post}, \mathsf{tx}) \overset{\tau_0}{\hookleftarrow} P$, check that $|\mathtt{PKI}| = |\mathcal{P}|$. If not, drop the message, else wait until round $\tau_1 \leq \tau_0 + \Delta$ (the exact value of τ_1 is determined by the adversary). Then check if:

1. The id is unique, i.e. for all $(t, \mathsf{tx}') \in \mathtt{TX}$, $\mathsf{tx}'.\mathsf{txid} \neq \mathsf{tx}.\mathsf{txid}$.
2. All the inputs are unspent and the witness satisfies all the output conditions, i.e. for each $(\mathsf{tid}, i) \in \mathsf{tx}.\mathsf{input}$, there exists $(t, \mathsf{tid}, i, \theta) \in \mathtt{UTXO}$ and $\theta.\varphi(\mathsf{tx}, t, \tau_1) = 1$.
3. All outputs are valid, i.e. for each $\theta \in \mathsf{tx}.\mathsf{output}$ it holds that $\theta.\mathsf{cash} > 0$ and $\theta.\varphi \in \mathcal{V}$.
4. The value of the outputs is not larger than the value of the inputs. More formally, let $I := \{\mathtt{utxo} := (t, \mathsf{tid}, i, \theta) \mid \mathtt{utxo} \in \mathtt{UTXO} \wedge (\mathsf{tid}, i) \in \mathsf{tx}.\mathsf{input}\}$, then $\sum_{\theta' \in \mathsf{tx}.\mathsf{output}} \theta'.\mathsf{cash} \leq \sum_{\mathtt{utxo} \in I} \mathtt{utxo}.\theta.\mathsf{cash}$
5. The absolute time-lock of the transaction has expired, i.e. $\mathsf{tx}.\mathsf{TimeLock} \leq \mathsf{now}$.

If all the above checks return true, add (τ_1, tx) to $\mathtt{TX}$, remove the spent outputs from $\mathtt{UTXO}$, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \setminus I$ and add the outputs of tx to $\mathtt{UTXO}$, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \cup \{(\tau_1, \mathsf{tx}.\mathsf{txid}, i, \theta_i)\}_{i \in [n]}$ for $(\theta_1, \ldots, \theta_n) := \mathsf{tx}.\mathsf{output}$. Else, ignore the message.

Read state: Upon $(\text{read}) \overset{\tau_0}{\hookleftarrow} X$, where X is any entity of the system, check that $|\mathtt{PKI}| = |\mathcal{P}|$. If not, drop the message, else $(\text{state}, \mathtt{PKI}, \mathtt{TX}) \overset{\tau_0}{\hookrightarrow} X$.

Figure B.1: Description of the global ledger functionality.

The GUC-security definition. Let π be a protocol with access to the global ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$ and the global clock $\widehat{\mathcal{F}}_{clock}$. The output of an environment $\mathcal{E}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input 1^λ and auxiliary input z is denoted as $EXEC^{\mathcal{L}(\Delta,\Sigma,\mathcal{V}),\widehat{\mathcal{F}}_{clock}}_{\pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. Let $\phi_{\mathcal{F}}$ be the ideal protocol for an ideal functionality $\mathcal{F}$ with access to the global ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$ and the global clock $\widehat{\mathcal{F}}_{clock}$. This means that $\phi_{\mathcal{F}}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. The output of an environment $\mathcal{E}$ interacting with a protocol $\phi_{\mathcal{F}}$ and a adversary $\mathcal{S}$ (sometimes also call *simulator*) on input 1^λ and auxiliary input z is denoted as $EXEC^{\mathcal{L}(\Delta,\Sigma,\mathcal{V}),\widehat{\mathcal{F}}_{clock}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

We are now ready to state our main security definition which, informally, says that if a protocol π UC-realizes an ideal functionality $\mathcal{F}$, then any attack that can be carried out against the real-world protocol π can also be carried out against the ideal protocol $\phi_{\mathcal{F}}$.

Definition 7. We say that a protocol π *UC-realizes an ideal functionality $\mathcal{F}$ with respect to a global ledger $\mathcal{L} := \mathcal{L}(\Delta, \Sigma, \mathcal{V})$* and a global clock $\widehat{\mathcal{F}}_{clock}$ if for every adversary $\mathcal{A}$ there

exists an adversary $\mathcal{S}$ such that we have

$$\left\{EXEC^{\mathcal{L},\widehat{\mathcal{F}}_{clock}}_{\pi,\mathcal{A},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{EXEC^{\mathcal{L},\widehat{\mathcal{F}}_{clock}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

(where "$\overset{c}{\approx}$" denotes computational indistinguishability of distribution ensembles, see, e.g., [Gol06]).

To simplify exposition, we omit the session identifiers *sid* and the sub-session identifiers *ssid*. Instead, we will use expressions like "message m is a reply to message m'". We believe that this approach improves readability.

B.2 Schnorr-based Adaptor Signature

In this section, we recall the Schnorr-based adaptor signature construction put forward by Poelstra [Poe] and formally prove that it satisfies our security definitions. Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order q and let $R_g \subseteq \mathbb{G} \times \mathbb{Z}_q$ be a relation defined as $R_g := \{(Y, y) \mid Y = g^y\}$. The adaptor signature construction is defined with respect to the Schnorr signature scheme Σ_{Sch} for the group $\mathbb{G}$ and the relation R_g. We implicitly assume that all algorithms of the scheme (and the adversary) are parameterized by public parameters $pp := (g, q)$ and have access to a random oracle $\mathcal{H}\colon \{0,1\}^* \to \mathbb{Z}_q$.

For completeness, let us briefly recall the Schnorr signature scheme $\Sigma_{\mathsf{Sch}} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. The key generation algorithm samples $x \leftarrow \mathbb{Z}_q$ uniformly at random and returns $X := g^x \in \mathbb{G}$ as the public key and x as the secret key. The signing algorithm on input a message $m \in \{0,1\}^*$ computes $r := \mathcal{H}(X \| g^k \| m) \in \mathbb{Z}_q$ and $s := k + rx \in \mathbb{Z}_q$, for a $k \leftarrow \mathbb{Z}_q$ chosen uniformly at random, and outputs a signature $\sigma := (r, s)$. The verification algorithm on input a message $m \in \{0,1\}^*$ and signature $(r, s) \in \mathbb{Z}_q \times \mathbb{Z}_q$, verifies that $r = \mathcal{H}(X \| g^s \cdot X^{-r} \| m)$.

To extend Schnorr signatures to an adaptor signature scheme, we need a method to produce pre-signatures that depend on the statement Y and reveal the corresponding witness y once the full signature is published. To this end, the r-component of a pre-signature is computed as $\mathcal{H}(X \| g^k Y \| m)$, and s is computed as in standard Schnorr. To adapt a pre-signature into a complete signature, we need to adjust the randomness in s to make it consistent with the randomness $k + y$ used in the r-component. This is done by adding y to s, where y is a value s.t. $g^y = Y$. Clearly, given s and the fixed s-component, we can then efficiently compute the witness y. We formally define the Schnorr-based adaptor signature scheme $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$ in Figure B.2.

Theorem 9. *If the Schnorr signature scheme Σ_{Sch} is* SUF–CMA*-secure and R_g is a hard relation, then $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$ from Figure B.2 is a secure adaptor signature scheme in the ROM.*

Remark 1. *We note that Σ_{Sch} is* SUF–CMA*-secure under the assumption that the discrete logarithm problem is hard [KMP16]. However, since we prove the* aEUF–CMA*-security of*

$\mathsf{pSign}_{\mathsf{sk}}(m, Y)$	$\mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma})$	$\mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$	$\mathsf{Adapt}(\tilde{\sigma}, y)$
$k \leftarrow_{\$} \mathbb{Z}_q$	$(r, \tilde{s}) := \tilde{\sigma}$	$(r, s) := \sigma$	$(r, \tilde{s}) := \tilde{\sigma}$
$r := \mathcal{H}(X \Vert g^k Y \Vert m)$	$r' := \mathcal{H}(\mathsf{pk} \Vert g^{\tilde{s}} \mathsf{pk}^{-r} Y \Vert m)$	$(\tilde{r}, \tilde{s}) := \tilde{\sigma}$	$s := \tilde{s} + y$
$\tilde{s} := k + r \cdot \mathsf{sk}$	**return** $(r = r')$	$y' := s - \tilde{s}$	**return** (r, s)
return $(r, \tilde{s})$		**if** $(Y, y') \in R$	
		then return y'	
		else return $\perp$	

Figure B.2: Schnorr-based adaptor signature scheme $\Xi_{R_g, \Sigma_{\mathsf{Sch}}}$.

$\Xi_{R_g, \Sigma_{\mathsf{Sch}}}$ by a reduction to SUF–CMA*-security of Σ_{Sch}, we state the* SUF–CMA*-security of Σ_{Sch} in Theorem 9.*

In order to prove Theorem 9, we reduce both the unforgeability and the witness extractability of the adaptor signature scheme to the strong unforgeability of the standard Schnorr signature scheme. We first provide a high-level overview of the main technical challenges and thereafter present the full proof.

Suppose there exists a PPT adversary $\mathcal{A}$ that wins aSigForge (resp. aWitExt) experiment, then we design a PPT adversary (also called the simulator) $\mathcal{S}$ that breaks the SUF–CMA security. The main technical challenge in both reductions is that $\mathcal{S}$ has to answer queries (m, Y) to $\mathcal{O}_{\mathrm{pS}}$ by $\mathcal{A}$. This has to be done with access to the Schnorr signing oracle but without knowledge of sk and the witness y. Thus, we need a method to "transform" full signatures into valid pre-signatures without knowing y, which seems to go against the aEUF–CMA-security (resp. witness extractability).

To address this difficulty, we will use the programmability of the random oracle. Concretely, upon a pre-sign query by $\mathcal{A}$ on some message m, the simulator forwards this message to its own signing oracle and sends the resulting full signature back to $\mathcal{A}$. To "convince" $\mathcal{A}$ that the reply looks like a valid pre-signature, we program the random oracle for RO queries made to verify the pre-signatures. This is possible since the pre-signature and signature verification differ only in the inputs to the hash function.

Finally, let us briefly explain why we need that the underlying signature scheme is strongly unforgeable. In the reduction, $\mathcal{S}$ needs to simulate a pre-signature on the target message m for which a successful $\mathcal{A}$ will later produce a forgery. As described above, this is achieved by querying the underlying Schnorr signature oracle on message m. When $\mathcal{A}$ returns a full signature for m as its forgery, $\mathcal{S}$ can only use this forgery to break the strong unforgeability of Schnorr.

We are now prepared to present the full proof of Theorem 9. As a first step we prove that our Schnorr adaptor signature scheme satisfies pre-signature adaptability. In fact, we prove a slightly stronger statement; namely, that any valid pre-signature adapts to a valid signature with probability 1.

Lemma 7 (Pre-signature adaptability)**.** *The Schnorr-based adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$ *satisfies pre-signature adaptability.*

Proof. Let us fix arbitrary $y \in \mathbb{Z}_q$, $m \in \{0,1\}^*$, $\mathsf{pk} \in \mathbb{G}$ and $(r, \tilde{s}) \in \mathbb{Z}_q \times \mathbb{Z}_q$. Let us define $Y := g^y$ and $s := \tilde{s} + y$. Assuming that $\mathsf{pVrfy}_{\mathsf{pk}}(m, Y; (r, \tilde{s})) = 1$, we have

$$\begin{aligned} r &= \mathcal{H}(\mathsf{pk} \| g^{\tilde{s}} \mathsf{pk}^{-r} Y \| m) \\ &= \mathcal{H}(\mathsf{pk} \| g^{\tilde{s}+y} \mathsf{pk}^{-r} \| m) \\ &= \mathcal{H}(\mathsf{pk} \| g^{s} \mathsf{pk}^{-r} \| m) \end{aligned}$$

which implies that $\mathsf{Vrfy}_{\mathsf{pk}}(m; (r, s)) = 1$. □

Lemma 8 (Pre-signature correctness)**.** *The Schnorr-based adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$ *satisfies pre-signature correctness.*

Proof. Let us fix arbitrary $x, y \in \mathbb{Z}_q$ and $m \in \{0,1\}^*$, and define $X := g^x$ and $Y := g^y$. For $\tilde{\sigma} = (r, \tilde{s}) \leftarrow \mathsf{pSign}_x(m, Y)$ it holds that $r = \mathcal{H}(X \| g^k \cdot Y \| m)$ and $\tilde{s} = k + rx$, for some $k \in \mathbb{Z}_q$. Since

$$\mathcal{H}(X \| g^{\tilde{s}} X^{-r} Y \| m) = \mathcal{H}(X \| g^{k+rx} g^{-xr} Y \| m) = r,$$

we have $\mathsf{pVrfy}_X(m, Y; \tilde{\sigma}) = 1$. By Lemma 7, this implies that $\mathsf{Vrfy}_X(m, Y; \sigma) = 1$ for $\sigma = (r, s) := (r, \tilde{s} + y) = \mathsf{Adapt}_X(\tilde{\sigma}, y)$. Finally,

$$\mathsf{Ext}((r, s), (r, \tilde{s}), Y) = s - \tilde{s} = (\tilde{s} + y) - \tilde{s} = y$$

which completes the proof.

□

Before we prove that the Schnorr-based adaptor signature scheme satisfies unforgeability, we make the following simple but useful observation.

Lemma 9. *For any* $\sigma := (r, s) \in \mathbb{Z}_q \times \mathbb{Z}_q$ *and any* $y \in \mathbb{Z}_q$ *it holds that*

$$\mathsf{Adapt}(\mathsf{Adapt}(\sigma, y), -y) = \sigma.$$

Proof. By definition of Adapt, for any $r, s, y \in \mathbb{Z}_q$ we have

$$\begin{aligned} \mathsf{Adapt}(\mathsf{Adapt}((r, s), y), -y) &= \mathsf{Adapt}((r, s + y), -y) \\ &= (r, s + y + (-y)) = (r, s) \end{aligned}$$

□

This lemma, in particular, implies that knowing a witness y one can not only adapt a valid pre-signature w.r.t. g^y into a valid signature but also the other way round.

Lemma 10 (aEUF–CMA security)**.** *Assuming that the Schnorr digital signature scheme* Σ_{Sch} *is* SUF–CMA*-secure and* R_g *is a hard relation, the adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$*, as defined in Figure B.2, is* aEUF–CMA *secure.*

Before we present the formal proof, let us give some intuition about the main ideas of the proof. Our goal is to reduce the unforgeability of the adaptor signature scheme to the strong unforgeability of the standard Schnorr signature scheme, i.e. we assume that there exists a PPT adversary $\mathcal{A}$ winning the aSigForge experiment and design a PPT adversary (also called the simulator) $\mathcal{S}$ winning the strongSigForge experiment. The main technical challenge in the reduction is the simulation of pre-sign queries. Since the reduction has access to the Schnorr signing oracle, it may ask for a full signature on the given message. However, it is not immediately clear how this helps to produce a pre-signature w.r.t. a given statement *without* knowing a witness. In fact, this might seem to go against the intuition that it is infeasible to transform a valid pre-signature to a full signature and vice versa without knowing a corresponding witness.

We make use of the fact that the reduction simulates not only the sign and pre-sign queries but also the queries to the random oracle. The main trick in simulating pre-sign queries is to simply forward the full signature to the adversary and "convince" him that it is a valid pre-signature. In more detail, we program the random oracle such that queries made during pre-signature verification are answered as if they were queries made during signature verification and vice versa. This is possible since the pre-signature and signature verification differ only in the string being hashed.

Let us emphasize that no oracle programming is needed for the pre-signature on the forgery message m. This is because the statement/witness pair (Y, y) is chosen by the reduction simulating the aSigForge experiment. The reduction can hence ask the Schnorr signing oracle for a signature σ on the message m and adapt it into a valid pre-signature $\tilde{\sigma}$ itself by executing $\mathsf{Adapt}(\sigma, -y)$. Now if the adversary outputs a valid signature σ', there are two options. Either $\sigma' \neq \sigma$, in which case the reduction learns a valid strongSigForge forgery, or $\sigma' = \sigma$, in which case the reductions failed. However, the latter case happens only with negligible probability since it implies that the adversary, given statement Y, found a witness y and hence broke the hardness of the relation R_g.

Proof. We prove the lemma by defining several game hops.

Game G_0: This game, formally defined in Figure B.3, corresponds to the original aSigForge, where the adversary $\mathcal{A}$ has to come up with a valid forgery for a message m of his choice while having access to pre-sign oracle $\mathcal{O}_{\mathrm{pS}}$ and sign oracle $\mathcal{O}_{\mathrm{S}}$. Since we are in the ROM, the adversary (as well as all the algorithms of the scheme) has additionally access to a random oracle $\mathcal{H}$.

$$\Pr[G_0 = 1] = \Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R_g,\Sigma_{\mathsf{Sch}}}}(\lambda) = 1]$$

G_0

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk}, Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$7 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 :$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

Figure B.3: The formal definition of game G_0.

Game G_1: This game, formally defined in Figure B.4, works exactly as G_0 with the following exception. When the adversary outputs a forgery σ, the game checks if completing the pre-signature $\tilde{\sigma}$ using the secret value y results in σ. If yes, the game aborts.

G_1

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk}, Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$7 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma$
$9 :$ Abort
$10 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$11 :$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

Figure B.4: The formal definition of G_1.

Claim: Let Bad_1 be the event that G_1 aborts. Then $\Pr[\mathsf{Bad}_1] \leq \nu_1(\lambda)$, where ν_1 is a negligible function in n.

Proof: We prove this claim using a reduction to the hardness of the relation R_g. More concretely, we construct a simulator $\mathcal{S}$ breaking the hardness the relation assuming he has access to an adversary $\mathcal{A}$ that causes G_1 to abort with non-negligible probability. The simulator gets a challenge Y^*, upon which it generates a key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$ in

order to simulate $\mathcal{A}$'s queries to the oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$. This simulation of the oracles works as described in $\boldsymbol{G_1}$. Eventually, upon receiving the challenge message m from $\mathcal{A}$, $\mathcal{S}$ computes a pre-signature $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y^*)$ and returns the pair $(\tilde{\sigma}, Y^*)$ to the adversary who outputs a forgery σ. Assuming that Bad_1 happened (i.e. $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma$), we know that due to the correctness property, the simulator can extract y^* by executing $\mathsf{Ext}(\sigma, \tilde{\sigma}, Y^*)$ to obtain a valid statement/witness pair for the relation R_g, i.e. $(Y^*, y^*) \in R_g$.

First, we note that the view of $\mathcal{A}$ is indistinguishable from his view in $\boldsymbol{G_1}$, since the challenge Y^* is an instance of the hard relation R_g and hence equally distributed to the public output of GenR. Hence the probability of $\mathcal{S}$ breaking the hardness of the relation is equal to the probability of the Bad_1 event. By our assumption, this is non-negligible which is in contradiction with the hardness of R_g. ■

Since games $\boldsymbol{G_1}$ and $\boldsymbol{G_0}$ are equivalent except if event Bad_1 occurs, it holds that $\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \nu_1(\lambda)$.

Game $\boldsymbol{G_2}$: This game, formally defined in Figure B.5, behaves like the previous game with the only differences being in the $\mathcal{O}_{\mathrm{pS}}$ oracle. In this game, the $\mathcal{O}_{\mathrm{pS}}$ oracle makes a copy of the list H before executing the algorithm $\mathsf{pSign}_{\mathsf{sk}}$. Afterwards it extracts the randomness used during the $\mathsf{pSign}_{\mathsf{sk}}$ algorithm, and checks if before the execution of the signing algorithm, a query of the form $\mathsf{pk}\|K\|m$ or $\mathsf{pk}\|K \cdot Y\|m$ was made to $\mathcal{H}$ by checking if $H'[\mathsf{pk}\|K\|m] \neq \bot$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$. If so the game aborts.

$\boldsymbol{G_2}$

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk}, Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$7 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma$
$9 :$ Abort
$10 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$11 :$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_{\$} \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1 : H' := H$
$2 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$3 : (r, s) := \tilde{\sigma}$
$4 : K := g^s \cdot \mathsf{pk}^{-r}$
$5 :$ **if** $(H'[\mathsf{pk}\|K\|m] \neq \bot$
$6 :$ $\vee H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot)$
$7 :$ Abort
$8 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$9 :$ **return** $\tilde{\sigma}$

Figure B.5: The formal definition of $\boldsymbol{G_2}$.

Claim: Let Bad_2 be the event that $\boldsymbol{G_2}$ aborts in $\mathcal{O}_{\mathrm{pS}}$. Then $\Pr[\mathsf{Bad}_2] \leq \nu_2(\lambda)$, where ν_2 is a negligible function in n.

Proof: We first recall that $\mathsf{pSign}_{\mathsf{sk}}$ and $\mathsf{Sign}_{\mathsf{sk}}$ compute $K = g^k$ by choosing k uniformly at random from $\mathbb{Z}_q$. Since $\mathcal{A}$ is PPT, the number of queries it can make to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and

$\mathcal{O}_{\mathrm{pS}}$ is also polynomially bounded. Let l_1, l_2, l_3 be the number of queries made to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and $\mathcal{O}_{\mathrm{pS}}$ respectively, then we have:

$$\begin{aligned}\Pr[\mathsf{Bad}_2] &= \Pr[H'[\mathsf{pk}\|K\|m] \neq \perp \vee H'[\mathsf{pk}\|K \cdot Y\|m] \neq \perp] \\ &\leq 2\frac{l_1 + l_2 + l_3}{q} =: \nu_2(n)\end{aligned}$$

Since l_1, l_2, l_3 are polynomial in n, ν_2 is a negligible function. ■

Since games $\boldsymbol{G_2}$ and $\boldsymbol{G_1}$ are equivalent except if event Bad_2 occurs, it holds that $\Pr[G_1 = 1] \leq \Pr[G_2 = 1] + \nu_2(\lambda)$.

Game $\boldsymbol{G_3}$: In this game, formally defined in Figure B.6, upon an $\mathcal{O}_{\mathrm{pS}}$ query, the game produces a valid full signature $\tilde{\sigma} = (r, s) = (\mathcal{H}(\mathsf{pk}\|K\|m), k + r\mathsf{sk})$ and adjusts the global list H as follows: It assigns the value stored at position $\mathsf{pk}\|K\|m$ to $H[\mathsf{pk}\|K \cdot Y\|m]$ and samples a fresh random value for $H[\mathsf{pk}\|K\|m]$. These changes make the full signature $\tilde{\sigma}$ "look like" a pre-signature to the adversary, since upon querying the random oracle on $\mathsf{pk}\|K \cdot Y\|m$, $\mathcal{A}$ obtains the value $H[\mathsf{pk}\|K\|m]$. The adversary can only notice the changes in this game, in case the random oracle has been previously queried on either $\mathsf{pk}\|K\|m$ or $\mathsf{pk}\|K \cdot Y\|m$. This case has been captured in the previous game and hence it holds that $\Pr[G_2 = 1] = [G_3 = 1]$.

$\boldsymbol{G_3}$

$1: \mathcal{Q} := \emptyset$
$2: H := [\perp]$
$3: (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4: (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5: (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk}, Y)$
$6: \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$7: \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8:$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma$
$9:$ Abort
$10: b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$11:$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1: \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3:$ **return** σ

$\mathcal{H}(x)$

$1:$ **if** $H[x] = \perp$
$2:$ $H[x] \leftarrow_{\$} \mathbb{Z}_q$
$3:$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1: H' := H$
$2: \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$3: (r, s) := \tilde{\sigma}$
$4: K := g^s \cdot \mathsf{pk}^{-r}$
$5:$ **if** $(H'[\mathsf{pk}\|K\|m] \neq \perp$
$6:$ $\vee H'[\mathsf{pk}\|K \cdot Y\|m] \neq \perp)$
$7:$ Abort
$8: x := \mathsf{pk}\|K\|m$
$9: H[\mathsf{pk}\|K \cdot Y\|m] := H[x]$
$10: H[x] \leftarrow_{\$} \mathbb{Z}_q$
$11: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12:$ **return** $\tilde{\sigma}$

Figure B.6: The formal definition of $\boldsymbol{G_3}$.

Game $\boldsymbol{G_4}$: In this game, formally defined in Figure B.6, the pre-signature generated upon $\mathcal{A}$ outputting the message m is created by modifying a full signature to a pre-signature. In other words upon receiving the full signature $\sigma = (r, s)$, where $s = k + xr$ and $r = \mathcal{H}(g^x\|g^k\|m)$ and given the pair (Y, y), the game can modify the signature to the pre-signature by setting $\tilde{\sigma} = \mathsf{Adapt}(\sigma, -y)$. One way to see this transformation is that k is modified to $k' = k - y$.

G_4

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk}, Y)$
$6 : \sigma' \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$7 : (r, s) := \sigma'$
$8 : \tilde{\sigma} := \mathsf{Adapt}(\sigma, -y)$
$9 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$10 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma$
$11 :$ Abort
$12 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$13 :$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : H' := H$
$2 : \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$3 : (r, s) := \tilde{\sigma}$
$4 : K := g^s \cdot \mathsf{pk}^{-r}$
$5 :$ **if** $(H'[\mathsf{pk} \| K \| m] \neq \bot$
$6 :$ $\vee\, H'[\mathsf{pk} \| K \cdot Y \| m] \neq \bot)$
$7 :$ Abort
$8 : x := \mathsf{pk} \| K \| m$
$9 : H[\mathsf{pk} \| K \cdot Y \| m] := H[x]$
$10 : H[x] \leftarrow_\$ \mathbb{Z}_q$
$11 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12 :$ **return** $\tilde{\sigma}$

Figure B.7: The formal definition of G_4.

Since k is chosen uniformly at random and according to Lemma 9, the view of the adversary is identical in this game and the previous game and hence it holds that $\Pr[G_3 = 1] = [G_4 = 1]$.

Having shown that the transition from the original aSigForge game (game G_0) to game G_4 is indistinguishable, it remains to show that there exists a simulator that perfectly simulates G_4 and uses $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ to win the strongSigForge game. In the following, we concisely describe how the simulator answers oracle queries. The formal description of the simulator can be found in Figure B.8.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{S}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{Sch}}$ and forwards its response to $\mathcal{A}$.

Random Oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}$ on input x, if $H[x] = \bot$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{Sch}}(x)$, otherwise the simulator returns $H[x]$.

Pre-Signing queries:

1. Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{pS}$ on input (m, Y), $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{Sch}}$ and receives the signature $\tilde{\sigma} = (r, s)$ where $r = \mathcal{H}^{\mathsf{Sch}}(\mathsf{pk} \| K \| m)$.
2. If $\mathcal{H}$ has been previously queried on the input $(\mathsf{pk} \| K \| m)$ or $(\mathsf{pk} \| K \cdot Y \| m)$, $\mathcal{S}$ aborts.
3. $\mathcal{S}$ programs the random oracle $\mathcal{H}$ such that queries of $\mathcal{A}$ on the input $\mathsf{pk} \| K \cdot Y \| m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk} \| K \| m)$ and queries on the input $\mathsf{pk} \| K \| m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk} \| K \cdot Y \| m)$.

$\mathcal{S}^{\mathsf{Sign}^{\mathsf{Sch}},\mathcal{H}^{\mathsf{Sch}}}(\mathsf{pk})$

$1: \mathcal{Q} := \emptyset$
$2: H := [\bot]$
$3: (Y, y) \leftarrow \mathsf{GenR}(1^n)$
$4: (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\mathsf{pk}, Y)$
$5: \sigma' := \mathsf{Sign}^{\mathsf{Sch}}(m)$
$6: (r, s) := \sigma'$
$7: \tilde{\sigma} := \mathsf{Adapt}(\sigma, -y)$
$8: \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$9:$ **return** (m, σ)

$\mathcal{O}_{\mathrm{S}}(m)$

$1: \sigma := \mathsf{Sign}^{\mathsf{Sch}}(m)$
$2: (r, s) := \sigma$
$3: K := g^s \cdot \mathsf{pk}^{-r}$
$4: x := \mathsf{pk} \| K \| m$
$5: H[x] := \mathcal{H}^{\mathsf{Sch}}(x)$
$6: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$7:$ **return** σ

$\mathcal{H}(x)$

$1:$ **if** $H[x] = \bot$
$2:$ $H[x] := \mathcal{H}^{\mathsf{Sch}}(x)$
$3:$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1: H' := H$
$2: \tilde{\sigma} := \mathsf{Sign}^{\mathsf{Sch}}(m)$
$3: (r, s) := \tilde{\sigma}$
$4: K := g^s \cdot \mathsf{pk}^{-r}$
$5:$ **if** $H'[\mathsf{pk} \| K \| m] \neq \bot$
$6:$ or $H'[\mathsf{pk} \| K \cdot Y \| m] \neq \bot$
$7:$ Abort
$8: x := \mathsf{pk} \| K \| m$
$9: H[\mathsf{pk} \| K \cdot Y \| m] := \mathcal{H}^{\mathsf{Sch}}(x)$
$10: H[x] := \mathcal{H}^{\mathsf{Sch}}(\mathsf{pk} \| K \cdot Y \| m)$
$11: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12:$ **return** $\tilde{\sigma}$

Figure B.8: The formal definition of the simulator.

4. The simulator returns $\tilde{\sigma}$ to $\mathcal{A}$.

Challenge Phase: 1. $\mathcal{S}$ chooses values $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$ and runs the adversary $\mathcal{A}_1$ on pk and Y.

2. Upon $\mathcal{A}_1$ outputting the message m as the challenge message, $\mathcal{S}$ queries the $\mathsf{Sign}^{\mathsf{Sch}}$ oracle on input m. Let $\sigma' = (r, s)$ be the response, then $\mathcal{S}$ runs $\mathcal{A}_2$ on $\tilde{\sigma} = (r, s - y)$.
3. Upon $\mathcal{A}_2$ outputting a forgery σ, the simulator outputs (m, σ) as its own forgery.

We emphasize that the main differences between the simulation and $\boldsymbol{G_4}$ are syntactical, namely instead of generating the public and secret keys and calculating the algorithm $\mathsf{Sign}_{\mathsf{sk}}$ and the random oracle $\mathcal{H}$, the simulator $\mathcal{S}$ uses its oracles $\mathsf{Sign}^{\mathsf{Sch}}$ and $\mathcal{H}^{\mathsf{Sch}}$. Therefore $\mathcal{S}$ perfectly simulates $\boldsymbol{G_4}$.

It remains to show that the forgery output by $\mathcal{A}$ can be used by the simulator to win the strongSigForge game.

Claim: (m, σ) constitutes a valid forgery in game strongSigForge.

Proof: In order to prove this claim, we have to show that the tuple (m, σ) has not been output by the oracle $\mathsf{Sign}^{\mathsf{Sch}}$ before. Note that the adversary $\mathcal{A}$ has not previously made a query on the challenge message m to either $\mathcal{O}_{\mathrm{pS}}$ or $\mathcal{O}_{\mathrm{S}}$. Hence, $\mathsf{Sign}^{\mathsf{Sch}}$ is only queried on m during the challenge phase. As shown in game $\boldsymbol{G_1}$ and according to Lemma 9, the adversary outputs a forgery σ which is equal to the signature σ' output by $\mathsf{Sign}^{\mathsf{Sch}}$

during the challenge phase only with negligible probability (in this case the simulation aborts). Hence, $\mathsf{Sign}^{\mathsf{Sch}}$ has never output σ on query m before and consequently (m, σ) constitutes a valid forgery for game strongSigForge. ■

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \nu_1(\lambda) + \nu_2(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain: $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{aSigForge}} \leq \mathsf{Adv}^{\mathcal{S}}_{\mathsf{strongSigForge}} + \nu_1(\lambda) + \nu_2(\lambda)$.

□

Lemma 11 (Witness Extractability)**.** *Assuming that Schnorr digital signature scheme* Σ_{Sch} *is* SUF–CMA*-secure and* R_g *is a hard relation, the adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{Sch}}}$ *as defined in Fig. B.2 is witness extractable.*

Proof. Before giving the formal proof, we first provide the main intuition. In general, this proof is very similar to the proof of Lemma 10. Our goal is to reduce the witness extractability of the adaptor signature scheme to the strong unforgeability of the standard Schnorr signature scheme. More concretely, under the assumption that there exists a PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ winning the aWitExt experiment, we design a PPT adversary $\mathcal{S}$ that wins the strongSigForge experiment.

The simulation of pre-sign queries is done exactly as in the proof of Lemma 10. However, unlike in the aSigForge experiment, in aWitExt $\mathcal{A}_1$ outputs the public value Y alongside the challenge message m, meaning that the game does not choose the pair (Y, y). Therefore, $\mathcal{S}$ does not learn the witness y and hence cannot transform a full signature to a pre-signature by executing $\mathsf{Adapt}(\sigma, -y)$. Fortunately, we can do this transformation without knowledge of y by using the same random oracle programmability as in the $\mathcal{O}_{\mathrm{pS}}$ oracle. More concretely, $\mathcal{S}$ can program the random oracle such that queries made during pre-signature verification are answered as if they were queries made during signature verification and vice versa. In other words, the values $\mathcal{H}(g^x \| K \| m)$ and $\mathcal{H}(g^x \| KY \| m)$ (where $K = g^k$, g^x and Y are known to the simulator) are swapped in the random oracle.

We note that it is not possible to program the random oracle if at least one of the values $g^x \| K \| m$ or $g^x \| KY \| m$ have already been queried to $\mathcal{H}$. However, since $\mathcal{A}$ is PPT, and k is chosen uniformly at random from $\mathbb{Z}_q$ (during the signing and pre-signing processes) where q is exponential in n, the probability that one of these values has previously been queried to $\mathcal{H}$ is negligible in the security parameter n.

□

Game $\boldsymbol{G_0}$: This game, formally defined in Figure B.9, corresponds to the original aWitExt, where the adversary $\mathcal{A}$ has to come up with a valid forgery for a message m of his choice such that extracting the secret value given the forgery and the pre-signature is not in relation with the corresponding public key. $\mathcal{A}$ has access to oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$, and $\mathcal{O}_{\mathrm{S}}$, and since we are in the random oracle model, we explicitly write the random oracle code $\mathcal{H}$.

G_0

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk})$
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$6 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7 : y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)$
$8 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 : b_2 := m \notin \mathcal{Q}$
$10 : b_3 := (Y, y') \notin R$
$11 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

Figure B.9: The formal definition of G_0.

Game G_1: This game, formally defined in Figure B.10, behaves like G_0 with the only differences being in the $\mathcal{O}_\mathrm{pS}$ oracle. First, a copy of the list H is stored before executing the algorithm $\mathsf{pSign}_{\mathsf{sk}}$ in the oracle $\mathcal{O}_\mathrm{pS}$. Upon computing the pre-signature, the game extracts the randomness used during the $\mathsf{pSign}_{\mathsf{sk}}$ algorithm, and checks if before the execution of the signing algorithm, a query of the form $\mathsf{pk}\|K\|m$ or $\mathsf{pk}\|K \cdot Y\|m$ was made to $\mathcal{H}$. This is done by checking if $H'[\mathsf{pk}\|K\|m] \neq \bot$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$. If so the game aborts.

G_1

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk})$
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$6 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7 : y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)$
$8 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 : b_2 := m \notin \mathcal{Q}$
$10 : b_3 := (Y, y') \notin R$
$11 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : H' := H$
$2 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$3 :$ parse $\tilde{\sigma}$ as (r, s)
$4 : K := g^s \cdot \mathsf{pk}^{-r}$
$5 :$ **if** $H'[\mathsf{pk}\|K\|m] \neq \bot$
$6 :$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$
$7 :$ Abort
$8 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$9 :$ **return** $\tilde{\sigma}$

Figure B.10: The formal definition of game G_0.

Claim: Let Bad_1 be the event that $\boldsymbol{G_1}$ aborts in $\mathcal{O}_{\mathrm{pS}}$, then $\Pr[\mathsf{Bad}_1] \leq \nu(\lambda)$, where ν is a negligible function in n.

Proof: We first recall that $\mathsf{pSign}_{\mathsf{sk}}$ and $\mathsf{Sign}_{\mathsf{sk}}$ compute $K = g^k$ by choosing k uniformly at random from $\mathbb{Z}_q$. Since $\mathcal{A}$ is PPT, the number of queries it can make to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and $\mathcal{O}_{\mathrm{pS}}$ are also polynomially bounded. Let l_1, l_2, l_3 be the number of queries made to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and $\mathcal{O}_{\mathrm{pS}}$ respectively, then we have:

$$\begin{aligned}\Pr[\mathsf{Bad}_1] = \Pr[&H'(\mathsf{pk}\|K\|m) \neq \bot \\ &\vee H'(\mathsf{pk}\|K \cdot Y\|m) \neq \bot] \\ &\leq 2\frac{l_1 + l_2 + l_3}{q} \leq \nu(n)\end{aligned}$$

■

Since games $\boldsymbol{G_1}$ and $\boldsymbol{G_0}$ are equivalent except if event Bad_1 occurs, it holds that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_1} = 1] + \nu_1(\lambda)$.

Game $\boldsymbol{G_2}$: In this game, formally defined in Figure B.11, upon an $\mathcal{O}_{\mathrm{pS}}$ query, the game produces a valid full signature such that $\tilde{\sigma} = (r, s) = (\mathcal{H}(\mathsf{pk}\|K\|m), k + r\mathsf{sk})$ and modifies the global list H as follows: It sets the value stored at position $\mathsf{pk}\|K\|m$ to $H[\mathsf{pk}\|K \cdot Y\|m]$ and samples a fresh random value for $H[\mathsf{pk}\|K\|m]$. These changes make the full signature $\tilde{\sigma}$ look like a pre-signature to the adversary, since upon querying the random oracle on $\mathsf{pk}\|K \cdot Y\|m$, $\mathcal{A}$ obtains the value $H[\mathsf{pk}\|K\|m]$. The adversary can only notice the changes in this game, in case the random oracle has been previously queried on either $\mathsf{pk}\|K\|m$ or $\mathsf{pk}\|K \cdot Y\|m$. This case has been captured in the previous game and hence it holds that $\Pr[\boldsymbol{G_1} = 1] = \Pr[\boldsymbol{G_2} = 1]$.

$\boldsymbol{G_2}$

$1: \mathcal{Q} := \emptyset$
$2: H := [\bot]$
$3: (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4: (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk})$
$5: \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$6: \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7: y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)$
$8: b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9: b_2 := m \notin \mathcal{Q}$
$10: b_3 := (Y, y') \notin R$
$11:$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1: \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3:$ **return** σ

$\mathcal{H}(x)$

$1:$ **if** $H[x] = \bot$
$2:$ $H[x] \leftarrow_{\$} \mathbb{Z}_q$
$3:$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1: H' := H$
$2: \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$3:$ parse $\tilde{\sigma}$ as (r, s)
$4: K := g^s \cdot \mathsf{pk}^{-r}$
$5:$ **if** $H'[\mathsf{pk}\|K\|m] \neq \bot$
$6:$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$
$7:$ Abort
$8: x := \mathsf{pk}\|K\|m$
$9: H[\mathsf{pk}\|K \cdot Y\|m] := H[x]$
$10: H[x] \leftarrow_{\$} \mathbb{Z}_q$
$11: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12:$ **return** $\tilde{\sigma}$

Figure B.11: The formal definition of game $\boldsymbol{G_2}$.

Game G_3: In this game, formally defined in Figure B.12, we apply the exact same changes made in game G_1 in oracle $\mathcal{O}_{\mathrm{pS}}$ to the challenge phase of the game. First, a copy of the list H is stored before executing the algorithm $\mathsf{pSign}_{\mathsf{sk}}$ during the challenge phase of the game. Upon computing the pre-signature, the game extracts the randomness used during the $\mathsf{pSign}_{\mathsf{sk}}$ algorithm, and checks if before the execution of the pre-signing algorithm, a query of the form $\mathsf{pk}\|K\|m$ or $\mathsf{pk}\|K \cdot Y\|m$ was made to $\mathcal{H}$. This is done by checking if $H'[\mathsf{pk}\|K\|m] \neq \bot$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$. If so the game aborts.

G_3

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk})$
$5 : H' := H$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
$7 :$ parse $\tilde{\sigma}$ as (r, s)
$8 : K := g^s \cdot \mathsf{pk}^{-r}$
$9 :$ **if** $H'[\mathsf{pk}\|K\|m] \neq \bot$
$10 :$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$
$11 :$ Abort
$12 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, Y, \mathsf{st})$
$13 : y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)$
$14 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$15 : b_2 := m \notin \mathcal{Q}$
$16 : b_3 := (Y, y') \notin R$
$17 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_{\$} \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1 : H' := H$
$2 : \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$3 :$ parse $\tilde{\sigma}$ as (r, s)
$4 : K := g^s \cdot \mathsf{pk}^{-r}$
$5 :$ **if** $H'[\mathsf{pk}\|K\|m] \neq \bot$
$6 :$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \bot$
$7 :$ Abort
$8 : x := \mathsf{pk}\|K\|m$
$9 : H[\mathsf{pk}\|K \cdot Y\|m] := H[x]$
$10 : H[x] \leftarrow_{\$} \mathbb{Z}_q$
$11 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12 :$ **return** $\tilde{\sigma}$

Figure B.12: The formal definition of game G_3.

Claim: Let Bad_2 be the event that G_2 aborts in $\mathrm{Game}_3(\lambda)$ during the challenge phase, then $\Pr[\mathsf{Bad}_2] \leq \nu(\lambda)$, where ν is a negligible function in n.

Proof: This proof is analogous to the proof of claim B.2. ■

Since games G_3 and G_2 are equivalent except if event Bad_2 occurs, it holds that $\Pr[G_2 = 1] \leq \Pr[G_3 = 1] + \nu(\lambda)$.

Game G_4: In this game, formally defined in Figure B.13, we apply the exact same changes made in game G_2 in oracle $\mathcal{O}_{\mathrm{pS}}$ to the challenge phase of the game. As explained before the adversary receives a full signature but by programming the random oracle, from $\mathcal{A}$'s point of view the signature looks like a pre-signature. It holds that $\Pr[G_4 = 1] = \Pr[G_3 = 1]$.

Having shown that the transition from the original aWitExt game (Game G_0) to Game

G_4

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk})$
$5 : H' := H$
$6 : \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$7 :$ parse $\tilde{\sigma}$ as (r, s)
$8 : K := g^s \cdot \mathsf{pk}^{-r}$
$9 :$ **if** $H'[\mathsf{pk} \| K \| m] \neq \bot$
$10 :$ or $H'[\mathsf{pk} \| K \cdot Y \| m] \neq \bot$
$11 :$ Abort
$12 : x := \mathsf{pk} \| K \| m$
$13 : H[\mathsf{pk} \| K \cdot Y \| m] := H[x]$
$14 : H[x] \leftarrow_\$ \mathbb{Z}_q$
$15 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, Y, \mathsf{st})$
$16 : y' := \mathsf{Ext}_{\mathsf{pk}}(\sigma, \tilde{\sigma}, Y)$
$17 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$18 : b_2 := m \notin \mathcal{Q}$
$19 : b_3 := (Y, y') \notin R$
$20 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, Y)$

$1 : H' := H$
$2 : \tilde{\sigma} \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$3 :$ parse $\tilde{\sigma}$ as (r, s)
$4 : K := g^s \cdot \mathsf{pk}^{-r}$
$5 :$ **if** $H'[\mathsf{pk} \| K \| m] \neq \bot$
$6 :$ or $H'[\mathsf{pk} \| K \cdot Y \| m] \neq \bot$
$7 :$ Abort
$8 : x := \mathsf{pk} \| K \| m$
$9 : H[\mathsf{pk} \| K \cdot Y \| m] := H[x]$
$10 : H[x] \leftarrow_\$ \mathbb{Z}_q$
$11 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$12 :$ **return** $\tilde{\sigma}$

Figure B.13: The formal definition of game G_4.

G_4 is indistinguishable, it remains to show that there exists a simulator that perfectly simulates G_4 and uses $\mathcal{A}$ to win the strongSigForge game. In the following, we concisely describe how the simulator answers oracle queries. The formal simulator code can be found in Figure B.14.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{S}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{Sch}}$ and forwards its response to $\mathcal{A}$.

Random Oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}$ on input x, if $H[x] = \bot$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{Sch}}(x)$, otherwise the simulator returns $H[x]$.

Pre-Signing queries:
1. Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{pS}$ on input (m, Y), $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{Sch}}$ and receives the signature $\tilde{\sigma} = (r, s)$ where $r = \mathcal{H}^{\mathsf{Sch}}(\mathsf{pk} \| K \| m)$.
2. If $\mathcal{H}$ has been previously queried on the input $(\mathsf{pk} \| K \| m)$ or $(\mathsf{pk} \| K \cdot Y \| m)$, $\mathcal{S}$ aborts.

3. $\mathcal{S}$ programs the random oracle $\mathcal{H}$ such that queries of $\mathcal{A}$ on the input $\mathsf{pk}\|K \cdot Y\|m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K\|m)$ and queries on the input $\mathsf{pk}\|K\|m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K \cdot Y\|m)$.
4. The simulator returns $\tilde{\sigma}$ to $\mathcal{A}$.

Challenge Phase:

1. Upon $\mathcal{A}$ outputting the message and public value (m, Y) as the challenge message, $\mathcal{S}$ queries the $\mathsf{Sign}^{\mathsf{Sch}}$ oracle on input m. Let $\sigma = (r, s)$ be the response where $r = \mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K\|m)$, then $\mathcal{S}$ again programs the random oracle $\mathcal{H}$ such that queries of $\mathcal{A}$ on the input $\mathsf{pk}\|K \cdot Y\|m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K\|m)$ and queries on the input $\mathsf{pk}\|K\|m$ are answered with the value of $\mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K \cdot Y\|m)$.
2. Upon $\mathcal{A}$ outputting a forgery σ, the simulator outputs (m, σ) as its own forgery.

$\mathcal{S}^{\mathsf{Sign}^{\mathsf{Sch}}, \mathcal{H}^{\mathsf{Sch}}}(\mathsf{pk})$

$1: \mathcal{Q} := \emptyset$
$2: H := [\perp]$
$3: (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4: (m, Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk})$
$5: H' := H$
$6: \tilde{\sigma} \leftarrow \mathsf{Sign}^{\mathsf{Sch}}(m)$
$7:$ parse $\tilde{\sigma}$ as (r, s)
$8: K := g^s \cdot \mathsf{pk}^{-r}$
$9:$ **if** $H'[\mathsf{pk}\|K\|m] \neq \perp$
$10:$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \perp$
$11:$ Abort
$12: x := \mathsf{pk}\|K\|m$
$13: H[\mathsf{pk}\|K \cdot Y\|m] \leftarrow_{\$} \mathcal{H}^{\mathsf{Sch}}(x)$
$14: H[x] \leftarrow_{\$} \mathcal{H}^{\mathsf{Sch}}(\mathsf{pk}\|K \cdot Y\|m)$
$15: \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, Y, \mathsf{st})$
$16:$ **return** (m, σ)

$\mathcal{O}_{\mathrm{S}}(m)$

$1: \sigma \leftarrow \mathsf{Sign}^{\mathsf{Sch}}(m)$
$2:$ parse σ as (r, s)
$3: K := g^s \cdot \mathsf{pk}^{-r}$
$4: x := \mathsf{pk}\|K\|m$
$5: H[x] \leftarrow \mathcal{H}^{\mathsf{Sch}}(x)$
$6: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$7:$ **return** σ

$\mathcal{H}(x)$

$1:$ **if** $H[x] = \perp$
$2:$ $H[x] \leftarrow_{\$} \mathcal{H}^{\mathsf{Sch}}(x)$
$3:$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

$1: H' := H$
$2: \sigma \leftarrow \mathsf{Sign}^{\mathsf{Sch}}(m)$
$3:$ parse $\tilde{\sigma}$ as (r, s)
$4: K := g^s \cdot \mathsf{pk}^{-r}$
$5:$ **if** $H'[\mathsf{pk}\|K\|m] \neq \perp$
$6:$ or $H'[\mathsf{pk}\|K \cdot Y\|m] \neq \perp$
$7:$ Abort
$8: x := \mathsf{pk}\|K\|m$
$9: y := \mathsf{pk}\|K \cdot Y\|m$
$10: H[y] \leftarrow_{\$} \mathcal{H}^{\mathsf{Sch}}(x)$
$11: H[x] \leftarrow_{\$} \mathcal{H}^{\mathsf{Sch}}(y)$
$12: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$13:$ **return** $\tilde{\sigma}$

Figure B.14: The formal definition of the simulator.

We emphasize that the main differences between the simulation and $\boldsymbol{G_4}$ are syntactical, namely instead of generating the public and secret keys and calculating the algorithm $\mathsf{Sign}_{\mathsf{sk}}$ and the random oracle $\mathcal{H}$, $\mathcal{S}$ uses its oracles $\mathsf{Sign}^{\mathsf{Sch}}$ and $\mathcal{H}^{\mathsf{Sch}}$.

It remains to show that the signature output by $\mathcal{A}$ can be used by the simulator to win the $\mathsf{strongSigForge}$ game.

Claim: (m, σ) constitutes a valid forgery in game $\mathsf{strongSigForge}$.

Proof: In order to prove this claim, we have to show that the tuple (m, σ) has not been output by the oracle $\mathsf{Sign}^{\mathsf{Sch}}$ before. Note that the adversary $\mathcal{A}$ has not previously made a query on the challenge message m to either $\mathcal{O}_{\mathrm{pS}}$ or $\mathcal{O}_{\mathrm{S}}$. Hence, $\mathsf{Sign}^{\mathsf{Sch}}$ is only queried on m during the challenge phase. If the adversary outputs a forgery σ which is equal to the signature $\tilde{\sigma}$ output by $\mathsf{Sign}^{\mathsf{Sch}}$ the adversary loses the game because this would not be a valid signature given the programmed random oracle. Hence, $\mathcal{A}$ must output a valid signature $\sigma \neq \tilde{\sigma}$ and $\mathsf{Sign}^{\mathsf{Sch}}$ has never output σ on query m before, consequently (m, σ) constitutes a valid forgery for game $\mathsf{strongSigForge}$. ■

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + 2\nu(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain: $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{aSigForge}} \leq \mathsf{Adv}^{\mathcal{S}}_{\mathsf{strongSigForge}} + 2\nu(\lambda)$.

B.3 Proof of the ECDSA-based Adaptor Signature

In Section IV of the paper, we presented our ECDSA-based adaptor signature scheme and explained the main ideas of our security proof. We now provide the formal proof of Theorem 2 which we recall here the following completeness.

Theorem 2 (restated)**.** *Assuming that the positive ECDSA signature scheme* Σ_{ECDSA} *is* $\mathsf{SUF\text{–}CMA}$*-secure and* R'_g *is a hard relation, the ECDSA-based adaptor signature scheme* $\Xi_{R_g, \Sigma_{\mathsf{ECDSA}}}$ *as defined in Figure 3.5 is secure in ROM.*

As a first step, we prove that our ECDSA-based adaptor signature scheme satisfies pre-signature adaptability. In fact, we prove a slightly stronger statement; namely, that any valid pre-signature adapts to a valid signature with probability 1.

Lemma 12 (Pre-signature adaptability)**.** *The ECDSA-based adaptor signature scheme* $\Xi_{R'_g, \Sigma_{\mathsf{ECDSA}}}$ *satisfies pre-signature adaptability.*

Proof. Let us fix arbitrary $(I_Y, y) \in R'_g$, $m \in \{0,1\}^*$, $X \in \mathbb{G}$ and $\tilde{\sigma} = (r, \tilde{s}, K, \pi) \in \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{G} \times \mathbb{G} \times \{0,1\}^*$. Let

$$\tilde{K} := g^{\mathcal{H}(m)\tilde{s}^{-1}} X^{r\tilde{s}^{-1}} \quad \text{and } r = f(K).$$

Assuming that $\mathsf{pVrfy}_X(m, I_Y; \tilde{\sigma}) = 1$, we know that there exists $k \in \mathbb{Z}_q$ s.t. $\tilde{K} = g^k$ and $K = Y^k$ for $(Y, \pi_Y) := I_Y$. By definition of Adapt, we know that $\mathsf{Adapt}(\tilde{\sigma}, y) = (r, s)$ for $s := \tilde{s} \cdot y^{-1}$. Hence, we have

$$\begin{aligned} f(g^{\mathcal{H}(m)s^{-1}} X^{rs^{-1}}) &= f((g^{\mathcal{H}(m)\tilde{s}^{-1}} X^{r\tilde{s}^{-1}})^y) \\ &= f(\tilde{K}^y) = f(K) = r. \end{aligned}$$

□

Lemma 13 (Pre-signature correctness)**.** *The ECDSA-based adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{ECDSA}}}$ *satisfies pre-signature correctness.*

Proof. Let us fix arbitrary $x, y \in \mathbb{Z}_q$ and $m \in \{0,1\}^*$, and define $X := g^x$, $Y := g^y$, $\pi_Y \leftarrow \mathsf{P}_g(Y)$ and $I_Y := (Y, \pi_Y)$. For $\tilde{\sigma} = (r, \tilde{s}, K, \pi) \leftarrow \mathsf{pSign}_x(m, I_Y)$ it holds that $\tilde{K} = g^k$, $K = Y^k$, $r = f(K)$ and $\tilde{s} = k^{-1}(\mathcal{H}(m) + rx)$. Set

$$\tilde{K} := g^{\mathcal{H}(m)\tilde{s}^{-1}} g^{r\tilde{s}^{-1}x} = g^k.$$

By correctness of NIZK_Y we know that $\mathsf{V}_Y((\tilde{K}, K), \pi) = 1$ and hence we also have that $\mathsf{pVrfy}_X(m, I_Y; \tilde{\sigma}) = 1$. By Lemma 12, this implies that $\mathsf{Vrfy}_X(m; \sigma) = 1$ for $\sigma = (r, s) := \mathsf{Adapt}(\tilde{\sigma}, y)$. By definition of Adapt, we know that $s = \tilde{s} \cdot y^{-1}$ and hence

$$\mathsf{Ext}((r, s), (r, \tilde{s}), I_Y) = s^{-1} \cdot \tilde{s} = (\tilde{s}^{-1} \cdot y^{-1}) \cdot \tilde{s} = y.$$

□

Lemma 14 (aEUF–CMA security)**.** *Assuming that the positive ECDSA signature scheme* Σ_{ECDSA} *is* SUF–CMA*-secure and* R'_g *is a hard relation, the adaptor signature scheme* $\Xi_{R_g,\Sigma_{\mathsf{ECDSA}}}$ *as defined in Figure 3.5 of the paper is* aEUF–CMA *secure.*

Proof. We prove unforgeability for the ECDSA-based adaptor signature scheme by reduction to strong unforgeability of positive ECDSA signatures. We consider an adversary $\mathcal{A}$ who plays the $\mathsf{aSigForge}$ game, then we build a simulator $\mathcal{S}$ who plays the strong unforgeability experiment for the ECDSA signature scheme and uses $\mathcal{A}$'s forgery in $\mathsf{aSigForge}$ to win its own experiment. $\mathcal{S}$ has access to the signing oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and the random oracle $\mathcal{H}^{\mathsf{ECDSA}}$, which it uses to simulate oracle queries for $\mathcal{A}$, namely random ($\mathcal{H}$), signing ($\mathcal{O}_{\mathrm{S}}$) and pre-signing ($\mathcal{O}_{\mathrm{pS}}$) queries.

The main challenges in the oracle simulations arise when simulating $\mathcal{O}_{\mathrm{pS}}$ queries, since $\mathcal{S}$ can only get full signatures from its own signing oracle and hence needs a way to transform those full signatures into pre-signatures for $\mathcal{A}$. In order to do so, the simulator faces two challenges, namely 1) $\mathcal{S}$ needs to learn the witness y for statement Y for which the pre-signature is supposed to be generated and 2) $\mathcal{S}$ needs to simulate the zero knowledge proof π which proves randomness consistency in the pre-signature.

More concretely, upon receiving a $\mathcal{O}_{\mathrm{pS}}$ query from $\mathcal{A}$ on input a message m and an instance $I_Y = (Y, \pi_Y)$, the simulator queries its Sign oracle to obtain a full signature on m. Further, $\mathcal{S}$ needs to learn a witness y, s.t. $Y = g^y$, in order to transform the full signature into a pre-signature for $\mathcal{A}$. We make use of the extractability property of the zero knowledge proof π_Y, in order to extract y and consequently transform a full signature into a valid pre-signature. Additionally, since a valid pre-signature contains a zero knowledge proof for L_{exp}, the simulator has to simulate this proof without knowledge of the corresponding witness. In order to do so, we make use of the zero knowledge property, which allows for simulation of a proof for a statement without knowing the corresponding witness.

Game G_0: This game, formally defined in Figure B.15, corresponds to the original aSigForge game, where the adversary $\mathcal{A}$ has to come up with a valid forgery for a message m of his choice, while having access to oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$. Since we are in the random oracle model, we explicitly write the random oracle code $\mathcal{H}$. We have $\Pr[G_0 = 1] = \Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R_g},\Sigma_{\mathsf{Sch}}}(\lambda) = 1]$.

$\boldsymbol{G_0}$

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\mathsf{pk}, I_Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$7 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma^*)$
$9 :$ **return** $(m \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_{\$} \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

Figure B.15: The formal definition of game $\boldsymbol{G_0}$.

Game G_1: This game, formally defined in Figure B.16, works exactly as $\boldsymbol{G_0}$ with the exception that upon the adversary outputting a forgery σ^*, the game checks if completing the pre-signature $\tilde{\sigma}$ using the witness y results in σ^*. In that case, the game aborts.

Claim: Let Bad_1 be the event that $\boldsymbol{G_1}$ aborts, then $\Pr[\mathsf{Bad}_1] \leq \nu(\lambda)$.

Proof: This proof is analogous to the proof of $\boldsymbol{G_1}$ in Lemma 10. ■

Since games $\boldsymbol{G_1}$ and $\boldsymbol{G_0}$ are equivalent except if event Bad_1 occurs, it holds that $\Pr[\boldsymbol{G_1} = 1] \leq \Pr[\boldsymbol{G_0} = 1] + \nu_1(\lambda)$, where ν_1 is a negligible function in λ.

Game G_2: This game, formally defined in Figure B.17, only applies changes to the $\mathcal{O}_{\mathrm{pS}}$ oracle as opposed to the previous game. Namely, during the $\mathcal{O}_{\mathrm{pS}}$ queries, this game extracts a witness y by executing the algorithm K on inputs the statement Y, the proof π_Y and the list of random oracle queries H. The game aborts, if for the extracted witness y it does not hold that $((Y, \pi_Y), y) \in R'_g$.

Claim: Let Bad_2 be the event that $\boldsymbol{G_2}$ aborts during an $\mathcal{O}_{\mathrm{pS}}$ execution, then it holds that $\Pr[\mathsf{Bad}_2] \leq \nu_2(\lambda)$ where ν_2 is a negligible function in λ.

Proof: According to the *online extractor* property of the zero knowledge proof, for a witness y extracted from a proof π_Y of statement Y such that $\mathsf{Vrfy}(Y, \pi_Y) = 1$, it holds that $((Y, \pi_Y), y) \in R'_g$ except with negligible probability in the security parameter. ■

G_1

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m^*, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\mathsf{pk}, I_Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m^*, I_Y)$
$7 : \sigma^* \leftarrow \mathcal{A}_2^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma^*$
$9 :$ Abort
$10 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
$11 :$ **return** $(m^* \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_S(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{O}_{pS}(m, I_Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

Figure B.16: The formal definition of game G_1.

Since games G_2 and G_1 are equivalent except if event Bad_2 occurs, it holds that $\Pr[G_2 = 1] \leq \Pr[G_1 = 1] + \nu_2(\lambda)$.

G_2

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m^*, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\mathsf{pk}, I_Y)$
$6 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m^*, I_Y)$
$7 : \sigma^* \leftarrow \mathcal{A}_2^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$8 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma^*$
$9 :$ Abort
$10 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
$11 :$ **return** $(m^* \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_S(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{O}_{pS}(m, I_Y)$

$1 :$ parse I_Y as (Y, π_Y)
$2 : y := \mathsf{K}(Y, \pi_Y, H)$
$3 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4 :$ **Abort**
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$6 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$7 :$ **return** $\tilde{\sigma}$

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \bot$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

Figure B.17: The formal definition of game G_2.

Game G_3: This game, formally defined in Figure B.18, extends the changes of the previous game to the $\mathcal{O}_{pS}$ oracle by first creating a valid full signature σ by executing the Sign algorithm and then converting σ into a pre-signature using the extracted witness y. Further, the game calculates the randomness $\tilde{K} = g^k$ and $K = \tilde{K}^{y^{-1}}$ from σ and simulates a zero knowledge proof π_S using $\tilde{K}$ and K.

Due to the *zero knowledge* property of the zero knowledge proof, the simulator can produce a proof π_{S} which is computationally indistinguishable from a proof $\pi \leftarrow P_{dh}((\tilde{K}, K), k)$. Hence, this game is indistinguishable from the previous game and it holds that $\Pr[\boldsymbol{G_3} = 1] \leq \Pr[\boldsymbol{G_2} = 1] + \nu_3(\lambda)$, where ν_3 is a negligible function in λ.

$\boldsymbol{G_3}$

$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$(I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$(m^*, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk}, I_Y)$
$\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m^*, I_Y)$
$\sigma^* \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
if $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma^*$
  Abort
$b := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
return $(m^* \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_{\mathrm{S}}(m)$

$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return σ

$\mathcal{H}(x)$

if $H[x] = \bot$
  $H[x] \leftarrow_{\$} \mathbb{Z}_q$
return $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$

parse I_Y as (Y, π_Y)
$y := \mathsf{K}(Y, \pi_Y, H)$
if $((Y, \pi_Y), y) \notin R'_g$
  Abort
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
parse σ as (r, s)
$\tilde{s} := s \cdot y$
$u := \mathcal{H}(m) \cdot s^{-1}$
$v := r \cdot s^{-1}$
$\tilde{K} := g^u X^v$
$K := \tilde{K}^{y^{-1}}$
$\pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(r, \tilde{s}, \tilde{K}, \pi_{\mathsf{S}})$

Figure B.18: The formal definition of game $\boldsymbol{G_3}$.

Game $\boldsymbol{G_4}$: In this game, which is formally defined in Figure B.19, upon receiving the challenge message m^* from $\mathcal{A}$, the game creates a full signature by executing the Sign algorithm and transforms the resulting signature into a pre-signature in the same way as in the previous game during the $\mathcal{O}_{\mathrm{pS}}$ execution. Hence, the same indistinguishability argument as in the previous game holds in this game as well and it holds that $\mathsf{Adv}^{\mathcal{A}}_{\boldsymbol{G_4}} \leq \mathsf{Adv}^{\mathcal{A}}_{\boldsymbol{G_3}} + \nu_3(\lambda)$, where ν_3 is a negligible function in λ.

Having shown that the transition from the original $\mathsf{aSigForge}$ game (Game $\boldsymbol{G_0}$) to Game $\boldsymbol{G_4}$ is indistinguishable, it remains to show that there exists a simulator that perfectly simulates $\boldsymbol{G_4}$ and uses $\mathcal{A}$ to win the $\mathsf{strongSigForge}$ game. In the following, we concisely describe how the simulator answers oracle queries. The formal description of the simulator can be found in Figure B.20.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_{\mathrm{S}}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and forwards its response to $\mathcal{A}$.

Random Oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}$ on input x, if $H[x] = \bot$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{ECDSA}}(x)$, otherwise the simulator returns $H[x]$.

G_4

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\perp]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$5 : (m^*, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk}, I_Y)$
$6 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m^*, I_Y)$
$7 :$ parse σ as (r, s)
$8 : \tilde{s} := s \cdot y$
$9 : u := \mathcal{H}(m^*) \cdot s^{-1}$
$10 : v := r \cdot s^{-1}$
$11 : \tilde{K} := g^u X^v$
$12 : K := \tilde{K}^{y^{-1}}$
$13 : \pi_\mathsf{S} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$14 : \tilde{\sigma} := (r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$
$15 : \sigma^* \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$16 :$ **if** $\mathsf{Adapt}(\tilde{\sigma}, y) = \sigma^*$
$17 :$ Abort
$18 : b := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
$19 :$ **return** $(m^* \notin \mathcal{Q} \wedge b)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H[x] = \perp$
$2 :$ $H[x] \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H[x]$

$\mathcal{O}_\mathrm{pS}(m, I_Y)$

$1 :$ parse I_Y as (Y, π_Y)
$2 : y := \mathsf{K}(Y, \pi_Y, H)$
$3 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4 :$ **Abort**
$5 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$6 :$ parse σ as (r, s)
$7 : \tilde{s} := s \cdot y$
$8 : u := \mathcal{H}(m) \cdot s^{-1}$
$9 : v := r \cdot s^{-1}$
$10 : \tilde{K} := g^u X^v$
$11 : K := \tilde{K}^{y^{-1}}$
$12 : \pi_\mathsf{S} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$13 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$14 :$ **return** $(r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$

Figure B.19: The formal definition of game G_4.

Pre-Signing queries: 1. Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{pS}$ on input (m, I_Y), the simulator extracts y using the extractability of NIZK, forwards m to oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and parses the signature that is generated as (r, s).

2. $\mathcal{S}$ generates a pre-signature from (r, s) by computing $\tilde{s} := s \cdot y$.

3. Finally, $\mathcal{S}$ simulates a zero knowledge proof π_S, proving that K and $\tilde{K}$ have the same exponent. The simulator outputs $(r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$.

Challenge phase: 1. $\mathcal{S}$ generates $(I_Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$ and runs $\mathcal{A}_1$ on I_Y

2. Upon $\mathcal{A}_1$ outputting the message m^* as the challenge message, $\mathcal{S}$ forwards m^* to the oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and parses the signature that is generated as (r, s).

3. The simulator generates the required pre-signature $\tilde{\sigma}$ in the same way as during $\mathcal{O}_\mathrm{pS}$ queries.

4. The simulator runs $\mathcal{A}_2$ on $\tilde{\sigma}$ and upon getting a forgery σ^*, the simulator outputs (m^*, σ^*) as its own forgery.

$\mathcal{S}^{\mathsf{Sign}^{\mathsf{ECDSA}},\mathcal{H}^{\mathsf{ECDSA}}}(\mathsf{pk})$

$1: \mathcal{Q} := \emptyset$
$2: H := [\bot]$
$3: (I_Y, y) \leftarrow \mathsf{GenR}(1^n)$
$4: (m^*, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\mathsf{pk}, I_Y)$
$5: \sigma \leftarrow \mathsf{Sign}^{\mathsf{ECDSA}}(m^*, I_Y)$
$6:$ parse σ as (r, s)
$7: \tilde{s} := s \cdot y$
$8: u := \mathcal{H}(m^*) \cdot s^{-1}$
$9: v := r \cdot s^{-1}$
$10: \tilde{K} := g^u X^v$
$11: K := \tilde{K}^{y^{-1}}$
$12: \pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$13: \tilde{\sigma} := (r, \tilde{s}, \tilde{K}, \pi_{\mathsf{S}})$
$14: \sigma^* \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$15:$ **return** (m^*, σ^*)

$\mathcal{O}_{\mathrm{S}}(m)$

$1: \sigma \leftarrow \mathsf{Sign}^{\mathsf{ECDSA}}(m)$
$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3:$ **return** σ

$\mathcal{H}(x)$

$1:$ **if** $H[x] = \bot$
$2:$ $H[x] \leftarrow_{\$} \mathcal{H}^{\mathsf{ECDSA}}(x)$
$3:$ **return** $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$

$1:$ parse I_Y as (Y, π_Y)
$2: y := \mathsf{K}(Y, \pi_Y, H)$
$3:$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4:$ **Abort**
$5: \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$6:$ parse σ as (r, s)
$7: \tilde{s} := s \cdot y$
$8: u := \mathcal{H}(m) \cdot s^{-1}$
$9: v := r \cdot s^{-1}$
$10: \tilde{K} := g^u X^v$
$11: K := \tilde{K}^{y^{-1}}$
$12: \pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$13: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$14:$ **return** $(r, \tilde{s}, \tilde{K}, \pi_{\mathsf{S}})$

Figure B.20: The formal definition of the simulator.

We emphasize that the main difference between the simulation and $\boldsymbol{G_4}$ are syntactical, namely instead of generating the public and secret keys and calculating the algorithm $\mathsf{Sign}_{\mathsf{sk}}$ and the random oracle $\mathcal{H}$, the simulator $\mathcal{S}$ uses its oracles $\mathsf{Sign}^{\mathsf{ECDSA}}$ and $\mathcal{H}^{\mathsf{ECDSA}}$.

It remains to show that the forgery output by $\mathcal{A}$ can be used by the simulator to win the strongSigForge game.

Claim: (m^*, σ^*) constitutes a valid forgery in game strongSigForge.

Proof: In order to prove this claim, we have to show that the tuple (m^*, σ^*) has not been output by the oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ before. Note that the adversary $\mathcal{A}$ has not previously made a query on the challenge message m^* to either $\mathcal{O}_{\mathrm{pS}}$ or $\mathcal{O}_{\mathrm{S}}$. Hence, $\mathsf{Sign}^{\mathsf{ECDSA}}$ is only queried on m^* during the challenge phase. As shown in game $\boldsymbol{G_1}$, the adversary outputs a forgery σ^* which is equal to the signature σ output by $\mathsf{Sign}^{\mathsf{ECDSA}}$ during the challenge phase only with negligible probability. Hence, $\mathsf{Sign}^{\mathsf{ECDSA}}$ has never output σ^* on query m^* before and consequently (m^*, σ^*) constitutes a valid forgery for game strongSigForge. ■

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \nu_1(\lambda) + \nu_2(\lambda) + 2\nu_3(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain:

$$\begin{aligned}\mathsf{Adv}^{\mathcal{A}}_{\mathsf{aSigForge}} &= \Pr[\boldsymbol{G_0} = 1] \\ &\leq \Pr[\boldsymbol{G_4}] + \nu_1(\lambda) + \nu_2(\lambda) + 2\nu_3(\lambda) \\ &\leq \mathsf{Adv}^{\mathcal{S}}_{\mathsf{strongSigForge}} + \nu_1(\lambda) + \nu_2(\lambda) + 2\nu_3(\lambda).\end{aligned}$$

□

Lemma 15 (Witness Extractability)**.** *Assuming that the positive ECDSA scheme* Σ_{pECDSA} *is* SUF–CMA*-secure and* R'_g *is a hard relation, the adaptor signature scheme* $\Xi_{R'_g,\Sigma_{\mathsf{pECDSA}}}$ *as defined in Figure 3.5 of the paper is witness extractable.*

Proof. Before providing the formal proof of witness extractability, we give the main intuition behind this proof. In general, this proof is very similar to the proof of lemma 14. Our goal is to reduce the witness extractability of $\Xi_{R'_g,\Sigma_{\mathsf{pECDSA}}}$ to the strong unforgeability of the positive ECDSA signature scheme. In other words, assuming that there exists a PPT adversary $\mathcal{A}$ who wins the aWitExt experiment, we design a PPT adversary $\mathcal{S}$ that wins the strongSigForge experiment.

During the reduction, the main challenge arises during the simulation of pre-sign queries. This simulation is done exactly as in the proof of lemma 14. However, unlike in the aSigForge experiment, in aWitExt, $\mathcal{A}$ outputs the statement I_Y for relation R'_g alongside the challenge message m^*, meaning that the game does not choose the pair (I_Y, y). Therefore, $\mathcal{S}$ does not learn the witness y and hence cannot transform a full signature to a pre-signature by computing $\tilde{s} := s \cdot y$. Fortunately, it is possible to extract y from the zero-knowledge proof embedded in I_Y. After extracting y, the same approach used in order to simulate the pre-sign queries can be taken here as well.

Game $\boldsymbol{G_0}$: This game, formally defined in Figure B.21, corresponds to the original aWitExt game, where the adversary $\mathcal{A}$ has to come up with a valid signature σ for a message m of his choice, a given pre-signature $\tilde{\sigma}$ and a given statement/witness pair $((Y, \pi_Y), y)$, while having access to oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$, such that $((Y, \pi_Y), \mathsf{Ext}(\sigma, \tilde{\sigma}, (Y, \pi_Y))) \notin R'_g$. Since we are in the random oracle model, we explicitly write the random oracle code $\mathcal{H}$. We have $\Pr[G_0 = 1] = \Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R_g},\Sigma_{\mathsf{Sch}}}(\lambda) = 1]$.

Game $\boldsymbol{G_1}$: This game, formally defined in Figure B.22, only applies changes to the $\mathcal{O}_{\mathrm{pS}}$ oracle as opposed to the previous game. Namely, during the $\mathcal{O}_{\mathrm{pS}}$ queries, this game extracts a witness y by executing the algorithm K on inputs the statement Y, the proof π_Y and the list of random oracle queries H. The game aborts, if for the extracted witness y it does not hold that $((Y, \pi_Y), y) \in R'_g$.

Claim: Let Bad_1 be the event that $\boldsymbol{G_1}$ aborts during an $\mathcal{O}_{\mathrm{pS}}$ execution, then it holds that $\Pr[\mathsf{Bad}_1] \leq \nu_1(\lambda)$, where ν_1 is a negligible function in λ.

Proof: According to the *online extractor* property of the zero knowledge proof, for a witness y extracted from a proof π_Y for statement Y such that $\mathsf{Vrfy}(Y, \pi_Y) = 1$, it holds

G_0

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\perp]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, I_Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\mathsf{pk})$
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$6 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7 : y' := \mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$
$8 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 : b_2 := m \notin \mathcal{Q}$
$10 : b_3 := (I_Y, y') \notin R'_g$
$11 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_S(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H(x) = \perp$
$2 :$ $H(x) \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H(x)$

$\mathcal{O}_{pS}(m, I_Y)$

$1 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** $\tilde{\sigma}$

Figure B.21: The formal definition of the game G_0.

that $((Y, \pi_Y Y), y) \in R'_g$ except with negligible probability. ■

Since games G_1 and G_0 are equivalent except if event Bad_1 occurs, it holds that $\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \nu_1(\lambda)$, where ν_1 is a negligible function in λ.

G_1

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\perp]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, I_Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\mathsf{pk})$
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$6 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_S, \mathcal{O}_{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7 : y' := \mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$
$8 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 : b_2 := m \notin \mathcal{Q}$
$10 : b_3 := (I_Y, y') \notin R'_g$
$11 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_S(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H(x) = \perp$
$2 :$ $H(x) \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H(x)$

$\mathcal{O}_{pS}(m, I_Y)$

$1 :$ parse I_Y as (Y, π_Y)
$2 :$ $y := \mathsf{K}(Y, \pi_Y, H)$
$3 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4 :$ **Abort**
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$6 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$7 :$ **return** $\tilde{\sigma}$

Figure B.22: The formal definition of the game G_1.

Game G_2: This game, formally defined in Figure B.23, extends the changes to $\mathcal{O}_{pS}$ from the previous game. In the $\mathcal{O}_{pS}$ execution, this game first creates a valid full signature σ by executing the Sign algorithm and converts σ into a pre-signature using the extracted

witness y. Further, the game calculates the randomness $\tilde{K} = g^k$ and $K = \tilde{K}^{y^{-1}}$ from σ and simulates a zero knowledge proof π_{S} using $\tilde{K}$ and K. Due to the *zero knowledge* property of the zero knowledge proof, the simulator can produce a proof π_{S} which is indistinguishable from a proof $\pi \leftarrow \mathsf{P}_{dh}((\tilde{K}, K), k)$. Hence, this game is indistinguishable from the previous game. It holds that $\Pr[\boldsymbol{G_1} = 1] \leq \Pr[\boldsymbol{G_2} = 1] + \nu_2(\lambda)$, where ν_2 is a negligible function in λ.

$\boldsymbol{G_2}$

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m, I_Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\mathsf{pk})$
$5 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$6 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathrm{S}}, \mathcal{O}_{\mathrm{pS}}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$7 : y' := \mathsf{Ext}(\sigma, \tilde{\sigma}, I_Y)$
$8 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m; \sigma)$
$9 : b_2 := m \notin \mathcal{Q}$
$10 : b_3 := (I_Y, y') \notin R'_g$
$11 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_{\mathrm{S}}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H(x) = \bot$
$2 : \ H(x) \leftarrow_{\$} \mathbb{Z}_q$
$3 :$ **return** $H(x)$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$

$1 :$ parse I_Y as (Y, π_Y)
$2 : y := \mathsf{K}(Y, \pi_Y, H)$
$3 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4 : \quad$ **Abort**
$5 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$6 :$ parse σ as (r, s)
$7 : \tilde{s} := s \cdot y$
$8 : u := \mathcal{H}(m) \cdot s^{-1}$
$9 : v := r \cdot s^{-1}$
$10 : \tilde{K} := g^u X^v$
$11 : K := \tilde{K}^{y^{-1}}$
$12 : \pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$13 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$14 :$ **return** $(r, \tilde{s}, \tilde{K}, \pi_{\mathsf{S}})$

Figure B.23: The formal definition of the game $\boldsymbol{G_2}$.

Game $\boldsymbol{G_3}$: In this game, formally defined in Figure B.24, we apply the exact same changes made in game $\boldsymbol{G_1}$ in oracle $\mathcal{O}_{\mathrm{pS}}$ to the challenge phase of the game. During the challenge phase, this game extracts a witness y by executing the algorithm K on inputs the statement Y, the proof π_Y and the list of random oracle queries H. The game aborts, if for the extracted witness y it does not hold that $((Y, \pi_Y), y) \in R'_g$.

Claim: Let Bad_2 be the event that $\boldsymbol{G_3}$ aborts during the challenge phase, then it holds that $\Pr[\mathsf{Bad}_2] \leq \nu_1(\lambda)$, where ν_1 is a negligible function in λ.

Proof: This proof is analogous to the proof of $\boldsymbol{G_1}$ in the proof of Lemma 15. ■

Since games $\boldsymbol{G_2}$ and $\boldsymbol{G_3}$ are equivalent except if event Bad_2 occurs, it holds that $\Pr[\boldsymbol{G_2} = 1] \leq \Pr[\boldsymbol{G_3} = 1] + \nu_1(\lambda)$, where ν_1 is a negligible function in λ.

Game $\boldsymbol{G_4}$: In this game, formally defined in Figure B.25, we apply the exact same changes made in game $\boldsymbol{G_2}$ in oracle $\mathcal{O}_{\mathrm{pS}}$ to the challenge phase of the game. In the challenge phase, this game first creates a valid full signature σ by executing the Sign algorithm and converts σ into a pre-signature using the extracted witness y. Further, the game calculates the randomness $\tilde{K} = g^k$ and $K = \tilde{K}^{y^{-1}}$ from σ and simulates a zero

G_3

$1 : \mathcal{Q} := \emptyset$
$2 : H := [\bot]$
$3 : (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$4 : (m^*, I_Y, \mathsf{st}) \leftarrow \mathcal{A}_1^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk})$
$5 :$ parse I_Y as (Y, π_Y)
$6 : y := \mathsf{K}(Y, \pi_Y, H)$
$7 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$8 :$ **Abort**
$9 : \tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, I_Y)$
$10 : \sigma \leftarrow \mathcal{A}_2^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$11 : y' := \mathsf{Ext}(\sigma^*, \tilde{\sigma}, I_Y)$
$12 : b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
$13 : b_2 := m^* \notin \mathcal{Q}$
$14 : b_3 := ((Y, \pi_Y), y') \notin R'_g$
$15 :$ **return** $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_\mathrm{S}(m)$

$1 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$2 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3 :$ **return** σ

$\mathcal{H}(x)$

$1 :$ **if** $H(x) = \bot$
$2 :$ $H(x) \leftarrow_\$ \mathbb{Z}_q$
$3 :$ **return** $H(x)$

$\mathcal{O}_\mathrm{pS}(m, I_Y)$

$1 :$ parse I_Y as (Y, π_Y)
$2 : y := \mathsf{K}(Y, \pi_Y, H)$
$3 :$ **if** $((Y, \pi_Y), y) \notin R'_g$
$4 :$ **Abort**
$5 : \sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$6 :$ parse σ as (r, s)
$7 : \tilde{s} := s \cdot y$
$8 : u := \mathcal{H}(m) \cdot s^{-1}$
$9 : v := r \cdot s^{-1}$
$10 : \tilde{K} := g^u X^v$
$11 : K := \tilde{K}^{y^{-1}}$
$12 : \pi_\mathsf{S} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$13 : \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$14 :$ **return** $(r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$

Figure B.24: The formal definition of the game $\boldsymbol{G_3}$.

knowledge proof π_S using $\tilde{K}$ and K. Due to the *zero knowledge* property of the zero knowledge proof, the simulator can produce a proof π_S which is indistinguishable from a proof $\pi \leftarrow \mathsf{P}_{dh}((\tilde{K}, K), k)$. Hence, this game is indistinguishable from the previous game. It holds that $\Pr[\boldsymbol{G_3} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \nu_3(\lambda)$, where ν_3 is a negligible function in λ.

Having shown that the transition from the original aWitExt game (Game $\boldsymbol{G_0}$) to Game $\boldsymbol{G_4}$ is indistinguishable, it remains to show that there exists a simulator that perfectly simulates $\boldsymbol{G_4}$ and uses $\mathcal{A}$ to win the strongSigForge game. In the following, we concisely describe how the simulator answers oracle queries. The simulator code can be found in Figure B.26.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{S}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and forwards its response to $\mathcal{A}$.

Random Oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}$ on input x, if $H[x] = \bot$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{ECDSA}}(x)$, otherwise the simulator returns $H[x]$.

Pre-Signing queries:
1. Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{pS}$ on input (m, I_Y), the simulator extracts y using the extractability of NIZK, forwards m to oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and parses the signature that is generated as (r, s).
2. $\mathcal{S}$ generates a pre-signature from (r, s) by computing $\tilde{s} := s \cdot y$.

G_4

$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
$(m^*, I_Y, \mathsf{st}) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\mathsf{pk})$
parse I_Y as (Y, π_Y)
$y := \mathsf{K}(Y, \pi_Y, H)$
if $((Y, \pi_Y), y) \notin R'_g$
  Abort
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m^*)$
parse σ as (r, s)
$\tilde{s} := s \cdot y$
$u := \mathcal{H}(m^*) \cdot s^{-1}$
$v := r \cdot s^{-1}$
$\tilde{K} := g^u X^v$
$K := \tilde{K}^{y^{-1}}$
$\pi_\mathsf{S} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$\tilde{\sigma} := (r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$
$\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}, \mathcal{O}_\mathrm{pS}, \mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
$y' := \mathsf{Ext}(\sigma^*, \tilde{\sigma}, I_Y)$
$b_1 := \mathsf{Vrfy}_{\mathsf{pk}}(m^*; \sigma^*)$
$b_2 := m^* \notin \mathcal{Q}$
$b_3 := ((Y, \pi_Y), y') \notin R'_g$
return $(b_1 \wedge b_2 \wedge b_3)$

$\mathcal{O}_\mathrm{S}(m)$

$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return σ

$\mathcal{H}(x)$

if $H(x) = \bot$
  $H(x) \leftarrow_\$ \mathbb{Z}_q$
return $H(x)$

$\mathcal{O}_\mathrm{pS}(m, I_Y)$

parse I_Y as (Y, π_Y)
$y := \mathsf{K}(Y, \pi_Y, H)$
if $((Y, \pi_Y), y) \notin R'_g$
  Abort
$\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
parse σ as (r, s)
$\tilde{s} := s \cdot y$
$u := \mathcal{H}(m) \cdot s^{-1}$
$v := r \cdot s^{-1}$
$\tilde{K} := g^u X^v$
$K := \tilde{K}^{y^{-1}}$
$\pi_\mathsf{S} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(r, \tilde{s}, \tilde{K}, \pi_\mathsf{S})$

Figure B.25: The formal definition of the game G_4.

3. Finally, $\mathcal{S}$ simulates a zero knowledge proof π_S, proving that it knows the exponent of K and $\tilde{K}$. The simulator outputs $(r, \tilde{s}, \tilde{K}, \pi)$.

Challenge phase: 1. Upon $\mathcal{A}$ outputting the message (m^*, I_Y) as the challenge message, $\mathcal{S}$ extracts y using the extractability of NIZK, forwards m^* to the oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ and parses the signature that is generated as (r, s).

2. The simulator generates the required pre-signature $\tilde{\sigma}$ in the same way as during $\mathcal{O}_\mathrm{pS}$ queries.

3. Upon $\mathcal{A}$ outputting a forgery σ, the simulator outputs (m^*, σ^*) as its own forgery.

We emphasize that the main differences between the simulation and G_4 are syntactical,

namely instead of generating the public and secret keys and calculating the algorithm $\mathsf{Sign}_{\mathsf{sk}}$ and the random oracle $\mathcal{H}$, the simulator $\mathcal{S}$ uses its oracles $\mathsf{Sign}^{\mathsf{ECDSA}}$ and $\mathcal{H}^{\mathsf{ECDSA}}$.

$\mathcal{S}^{\mathsf{Sign}^{\mathsf{ECDSA}},\mathcal{H}^{\mathsf{ECDSA}}}(\mathsf{pk})$

$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(m^*, I_Y, \mathsf{st}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\mathsf{pk})$
parse I_Y as (Y, π_Y)
$y := \mathsf{K}(Y, \pi_Y, H)$
if $((Y, \pi_Y), y) \notin R'_g$
  Abort
$\sigma \leftarrow \mathsf{Sign}^{\mathsf{ECDSA}}(m^*)$
parse σ as (r, s)
$\tilde{s} := s \cdot y$
$u := \mathcal{H}(m^*) \cdot s^{-1}$
$v := r \cdot s^{-1}$
$\tilde{K} := g^u X^v$
$K := \tilde{K}^{y^{-1}}$
$\pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$\tilde{\sigma} := (r, \tilde{s}, K, \pi_{\mathsf{S}})$
$\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}},\mathcal{H}}(\tilde{\sigma}, \mathsf{st})$
return (m^*, σ^*)

$\mathcal{O}_{\mathrm{S}}(m)$

$\sigma \leftarrow \mathsf{Sign}^{\mathsf{ECDSA}}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return σ

$\mathcal{H}(x)$

if $H(x) = \bot$
  $H(x) \leftarrow_{\$} \mathcal{H}^{\mathsf{ECDSA}}(x)$
return $H(x)$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$

parse I_Y as (Y, π_Y)
$y := \mathsf{K}(Y, \pi_Y, H)$
if $((Y, \pi_Y), y) \notin R'_g$
  Abort
$\sigma \leftarrow \mathsf{Sign}^{\mathsf{ECDSA}}(m)$
parse σ as (r, s)
$\tilde{s} := s \cdot y$
$u := \mathcal{H}(m) \cdot s^{-1}$
$v := r \cdot s^{-1}$
$\tilde{K} := g^u X^v$
$K := \tilde{K}^{y^{-1}}$
$\pi_{\mathsf{S}} \leftarrow \mathsf{S}((\tilde{K}, K), 1)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(r, \tilde{s}, K, \pi_{\mathsf{S}})$

Figure B.26: The formal definition of the game $\boldsymbol{G_3}$.

It remains to show that the signature output by $\mathcal{A}$ can be used by the simulator to win the $\mathsf{strongSigForge}$ game.

Claim: (m^*, σ^*) constitutes a valid forgery in game $\mathsf{strongSigForge}$.

Proof: In order to prove this claim, we have to show that the tuple (m^*, σ^*) has not been output by the oracle $\mathsf{Sign}^{\mathsf{ECDSA}}$ before. Note that the adversary $\mathcal{A}$ has not previously made a query on the challenge message m^* to either $\mathcal{O}_{\mathrm{pS}}$ or $\mathcal{O}_{\mathrm{S}}$. Hence, $\mathsf{Sign}^{\mathsf{ECDSA}}$ is only queried on m^* during the challenge phase. If the adversary outputs a forgery σ^* which is equal to the signature σ output by $\mathsf{Sign}^{\mathsf{ECDSA}}$ during the challenge phase, the extracted y would be in relation with the given public value I_Y. Hence, $\mathsf{Sign}^{\mathsf{ECDSA}}$ has never output σ^* on query m^* before and consequently (m^*, σ^*) constitutes a valid forgery for game $\mathsf{strongSigForge}$. ■

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + 2\nu_1(\lambda) + \nu_2(\lambda) + \nu_3(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{aWitExt}} &= \Pr[\boldsymbol{G_0} = 1] \\ &\leq \Pr[\boldsymbol{G_4} = 1] + 2\nu_1(\lambda) + \nu_2(\lambda) + \nu_3(\lambda) \\ &\leq \mathsf{Adv}^{\mathcal{S}}_{\mathsf{strongSigForge}} + 2\nu_1(\lambda) + \nu_2(\lambda) + \nu_3(\lambda)\end{aligned}$$

which concludes the proof. □

B.4 Pre-signature unforgeability

As mentioned in Section 3.5, the definition of aEUF–CMA does not explicitly state that pre-signatures are unforgeable. In this section, we prove that pre-signature unforgeability is, however, implied by Definition 3. In order to do so, let us first define pre-signature unforgeability formally.

Definition 8 (Pre-signature unforgeability)**.** An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfied *pre-signature unforgeability under chosen message attack (*pEUF–CMA *for short)* if for every PPT adversary $\mathcal{A}$ there exists a negligible function ν such that $\Pr[\mathsf{pSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda) = 1] \leq \nu(\lambda)$, where the experiment $\mathsf{pSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows:

$\mathsf{pSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$

1 : $\mathcal{Q} := \emptyset, (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
2 : $(Y, y) \leftarrow \mathsf{GenR}(1^n)$
3 : $(m, \tilde{\sigma}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk}, Y)$
4 : **return** $\left(m \notin \mathcal{Q} \wedge \mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma})\right)$

$\mathcal{O}_{\mathrm{S}}(m)$

1 : $\sigma \leftarrow \mathsf{Sign}_{\mathsf{sk}}(m)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** σ

$\mathcal{O}_{\mathrm{pS}}(m, Y)$

1 : $\tilde{\sigma} \leftarrow \mathsf{pSign}_{\mathsf{sk}}(m, Y)$
2 : $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
3 : **return** $\tilde{\sigma}$

Lemma 16. *If an adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies* aEUF–CMA *and pre-signature adaptability, then it also satisfies* pEUF–CMA*.*

Proof. Let $\mathcal{A}$ be a PPT adversary winning the pSigForge game with non-negligible probability. We construct an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to win the aSigForge game as follows:

Challenge phase:

1. Upon receiving a public key pk and a statement $Y \in L_R$ from the challenger, generate a statement/witness pair $(Y', y') \leftarrow \mathsf{GenR}(1^n)$.
2. Run the adversary $\mathcal{A}$ on pk and Y' to obtain $(m, \tilde{\sigma}')$.
3. Compute $\sigma' := \mathsf{Adapt}_{\mathsf{pk}}(\tilde{\sigma}', y')$
4. Output m to the challenger to obtain $\tilde{\sigma}$.

5. Return (m, σ') as a valid forgery.

Signing queries: If $\mathcal{A}$ makes a signing query, forward to request to $\mathcal{O}_{\mathrm{S}}$ and relay the answer.

Pre-Signing queries: If $\mathcal{A}$ makes a pre-signing query, forward to request to $\mathcal{O}_{\mathrm{pS}}$ and relay the answer.

Random Oracle queries: If $\mathcal{A}$ makes a query to the random oracle, forward to request to $\mathcal{H}$ and relay the answer.

□

It is easy to see that $\mathcal{B}$ perfectly simulates the $\mathsf{pSigForge}$ game to $\mathcal{A}$ and that $\mathcal{B}$ is a PPT algorithm. If $(m, \tilde{\sigma}')$ is a valid forgery, then $\mathsf{pVrfy}(m, Y'; \tilde{\sigma}) = 1$ and $\mathcal{A}$ did not query the signing or the pre-signing oracle on m. This implies that $m \notin \mathcal{Q}$. Moreover, pre-signature adaptability guarantees that $\sigma' := \mathsf{Adapt}_{\mathsf{pk}}(\tilde{\sigma}', y')$ is a valid signature on m. Hence (m, σ') is a successful forgery. To conclude, if $\mathcal{A}$ outputs a valid forgery, then so does $\mathcal{B}$. Hence, the success probability of $\mathcal{B}$ is non-negligible which completes the proof.

B.5 Additional material to generalized channel protocol

We now formally describe the protocol for generalized channels Π_L described at high level in Section 3.6 of the paper. The protocol internally uses a secure adaptor signature scheme $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ for the ledger signature scheme Σ and a relation R. We assume that statement/witness pairs of R are public/secret key of Σ. More precisely, we assume there exists a function ToKey that takes as input a statement $Y \in L_R$ and outputs a public key pk. The function is s.t. the distribution of $(\mathsf{ToKey}(Y), y)$, for $(Y, y) \leftarrow \mathsf{GenR}$, is equal to the distributions of $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}$. We emphasize that both ECDSA and Schnorr-based adaptor signatures, that we presented in Appendix B.2 and Section 3.5.1, satisfy this condition (ECDSA, the ToKey simply drops the NIZK, for Schnorr ToKey is the identity function). We discuss how to modify our protocol if this assumption does not hold in Remark 2 below the formal protocol description. Before we present our protocols, we introduce some conventions.

We assume that each party $P \in \mathcal{P}$ maintains a set Γ^P of all open channels together with auxiliary information about the channel (such as the funding transaction, latest commit transaction, and corresponding revocation secret, etc.). In addition to the channel set, we assume that each party maintains a set Θ^P containing all revoked commit transactions and corresponding revocation secretes. Similarly to the formal description of the ideal functionality, we make use of an arrow notation for sending and receiving messages which was explained in Section 3.3. Moreover, our formal description excludes some natural checks an honest party should make. These checks are defined as a protocol wrapper in Appendix B.7. In the protocol description, we abbreviate $\mathtt{One\text{–}Sig}_{pk_1} \wedge \cdots \wedge \mathtt{One\text{–}Sig}_{\mathsf{pk}_n}$

as $\texttt{Multi–Sig}_{\mathsf{pk}_1,\ldots,\mathsf{pk}_n}$. Moreover, we denote the script verifying that at least t rounds have passed since the transaction was accepted by the blockchains as $\texttt{CheckRelative}_t$.

In order to distinguish between the communication between parties and input/outputs from/to the environment, we use lowercase letters for the former and uppercase typewriter type style for the latter. So for example "CREATE" denotes a message from the environment while "createInfo" denotes a protocol message. To avoid code repetition, we define the generation of the funding, commit, and split transactions as separate subprocedure, presented at the end of the protocol description. For the same reason, we define the force closure as a subprocedure as well.

Generalized channel protocol

Below, we abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Create

Party P upon $(\texttt{CREATE}, \gamma, \mathsf{tid}_P) \xleftarrow{t_0} \mathcal{E}$:

1. Set $\mathsf{id} := \gamma.\mathsf{id}$, generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_P, R_P, Y_P) \xhookrightarrow{t_0} Q$.
2. If $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_Q, R_Q, Y_Q) \xhookleftarrow{t_0+1} Q$, create:

$$[\mathsf{tx}^\mathsf{f}] := \mathsf{GenFund}((\mathsf{tid}_P, \mathsf{tid}_Q), \gamma)$$
$$[\mathtt{TX}_\mathsf{c}] := \mathtt{GenCommit}([\mathsf{tx}^\mathsf{f}], I_P, I_Q)$$
$$[\mathtt{TX}_\mathsf{s}] := \mathtt{GenSplit}([\mathtt{TX}_\mathsf{c}].\mathsf{txid} \| 1, \gamma.\mathsf{st})$$

   for $I_P := (\mathsf{pk}_P, R_P, Y_P)$, $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$. Else stop.
3. Compute $s_\mathsf{c}^P \leftarrow \mathsf{pSign}_{\mathsf{sk}_P}([\mathtt{TX}_\mathsf{c}], Y_Q)$, $s_\mathsf{s}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_\mathsf{s}])$ and send $(\mathsf{createCom}, \mathsf{id}, s_\mathsf{c}^P, s_\mathsf{s}^P) \xhookrightarrow{t_0+1} Q$.
4. If $(\mathsf{createCom}, \mathsf{id}, s_\mathsf{c}^Q, s_\mathsf{s}^Q) \xhookleftarrow{t_0+2} Q$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_\mathsf{c}], Y_P; s_\mathsf{c}^Q) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_\mathsf{s}]; s_\mathsf{s}^Q) = 1$, $s_\mathsf{f}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathsf{tx}^\mathsf{f}])$ and send $(\mathsf{createFund}, \mathsf{id}, s_\mathsf{f}^P) \xhookrightarrow{t_0+2} Q$. Else stop.
5. If $(\mathsf{createFund}, \mathsf{id}, s_\mathsf{f}^Q) \xhookleftarrow{t_0+3} Q$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\mathsf{tx}^\mathsf{f}]; s_\mathsf{f}^Q) = 1$, $\mathsf{tx}^\mathsf{f} := ([\mathsf{tx}^\mathsf{f}], \{s_\mathsf{f}^P, s_\mathsf{f}^Q\})$ and $(\mathsf{post}, \mathsf{tx}^\mathsf{f}) \xhookrightarrow{t_0+3} \mathcal{L}$. Else parse $(\theta_P, \theta_Q) := \gamma.\mathsf{st}$, create tx such that $\mathsf{tx.input} := \mathsf{tid}_P$, $\mathsf{tx.output} := \theta_P$, $\mathsf{tx}.w \leftarrow \mathsf{Sign}_{\mathsf{pk}_P}([\mathsf{tx}])$ and $(\mathsf{post}, \mathsf{tx}) \xhookrightarrow{t_0+3} \mathcal{L}$.
6. If tx^f is accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + 3 + \Delta$, set $\mathtt{TX}_\mathsf{c} := ([\mathtt{TX}_\mathsf{c}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_\mathsf{c}]), \mathsf{Adapt}(s_\mathsf{c}^Q, y_P)\})$, $\mathtt{TX}_\mathsf{s} := ([\mathtt{TX}_\mathsf{s}], \{s_\mathsf{s}^P, s_\mathsf{s}^Q\})$, store $\Gamma^P(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx}^\mathsf{f}, (\mathtt{TX}_\mathsf{c}, r_P, R_Q, Y_Q, s_\mathsf{c}^P), \mathtt{TX}_\mathsf{s})$ and $(\texttt{CREATED}, \mathsf{id}) \xhookrightarrow{t_1} \mathcal{E}$.

Update

Party P upon $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xleftarrow{t_0} \mathcal{E}$

1. Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookrightarrow{t_0} Q$.

Party Q upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookleftarrow{\tau_0} P$

2. Generate $(R_Q, r_Q) \leftarrow \mathsf{GenR}$ and $(Y_Q, y_Q) \leftarrow \mathsf{GenR}$.
3. Extract tx^{f} from $\Gamma^P(\mathsf{id})$ and

$$[\texttt{TX}_{\mathsf{c}}] := \texttt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_P, I_Q)$$
$$[\texttt{TX}_{\mathsf{s}}] := \texttt{GenSplit}([\texttt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta)$$

   where $I_P := (\mathsf{pk}_P, R_P, Y_P)$, $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$.
4. Sign $s_{\mathsf{s}}^Q \leftarrow \mathsf{Sign}_{\mathsf{sk}_Q}([\texttt{TX}_{\mathsf{s}}])$, send $(\text{updateInfo}, \mathsf{id}, R_Q, Y_Q, s_{\mathsf{s}}^Q) \xhookrightarrow{\tau_0} P$, $(\texttt{UPDATE\_REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \texttt{TX}_{\mathsf{s}}.\mathsf{txid}) \xhookrightarrow{\tau_0+1} \mathcal{E}$.

Party P upon $(\text{updateInfo}, \mathsf{id}, h_Q, Y_Q, s_{\mathsf{s}}^Q) \xhookleftarrow{t_0+2} Q$

5. Extract tx^{f} from $\Gamma^Q(\mathsf{id})$ and

$$[\texttt{TX}_{\mathsf{c}}] := \texttt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_P, I_Q)$$
$$[\texttt{TX}_{\mathsf{s}}] := \texttt{GenSplit}([\texttt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta),$$

   for $I_P := (\mathsf{pk}_P, R_P, Y_P)$ and $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$. If $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\texttt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^Q) = 1$, $(\texttt{SETUP}, \mathsf{id}, \texttt{TX}_{\mathsf{s}}.\mathsf{txid}) \xhookrightarrow{t_0+2} \mathcal{E}$. Else stop.
6. If $(\texttt{SETUP\_OK}, \mathsf{id}) \xhookleftarrow{t_1 \le t_0+2+t_{\mathsf{stp}}} \mathcal{E}$, compute $s_{\mathsf{c}}^P \leftarrow \mathsf{pSign}_{\mathsf{sk}_P}([\texttt{TX}_{\mathsf{c}}], Y_Q)$, $s_{\mathsf{s}}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\texttt{TX}_{\mathsf{s}}])$ and send $(\text{updateComP}, \mathsf{id}, s_{\mathsf{c}}^P, s_{\mathsf{s}}^P) \xhookrightarrow{t_1} Q$. Else stop.

Party Q

7. If $(\text{updateComP}, \mathsf{id}, s_{\mathsf{c}}^P, s_{\mathsf{s}}^P) \xhookleftarrow{\tau_1 \le \tau_0+2+t_{\mathsf{stp}}} P$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{c}}], Y_Q; s_{\mathsf{c}}^P) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^P) = 1$, output $(\texttt{SETUP\_OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{E}$. Else stop.
8. If $(\texttt{UPDATE\_OK}, \mathsf{id}) \xhookleftarrow{\tau_1} \mathcal{E}$, pre-sign $s_{\mathsf{c}}^Q \leftarrow \mathsf{pSign}([\texttt{TX}_{\mathsf{c}}], Y_P)$ and send $(\text{updateComQ}, \mathsf{id}, s_{\mathsf{c}}^Q) \xhookrightarrow{\tau_1} P$. Else send $(\text{updateNotOk}, \mathsf{id}, r_Q) \xhookrightarrow{\tau_1} P$ and stop.

Party P

9. In round $t_1 + 2$ distinguish the following cases:

- If $(\text{updateComQ}, \mathsf{id}, s_{\mathsf{c}}^{Q}) \xleftarrow{t_1+2} Q$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_{\mathsf{c}}], Y_P; s_{\mathsf{c}}^{Q}) = 1$, output $(\mathtt{UPDATE\text{–}OK}, \mathsf{id}) \xrightarrow{t_1+2} \mathcal{E}$.
- If $(\text{updateNotOk}, \mathsf{id}, r_Q) \xleftarrow{t_1+2} Q$, s.t. $(R_Q, r_Q) \in R$, add $\Theta^P(\mathsf{id}) := \Theta^P(\mathsf{id}) \cup ([\mathtt{TX}_{\mathsf{c}}], r_Q, Y_Q, s_{\mathsf{c}}^{P})$ and stop.
- Else, execute the procedure $\mathtt{L\text{–}ForceClose}^P(\mathsf{id})$ and stop.

10. If $(\mathtt{REVOKE}, \mathsf{id}) \xleftarrow{t_1+2} \mathcal{E}$, parse $\Gamma^P(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\mathtt{TX}}_{\mathsf{c}}, \bar{r}_P, \bar{R}_Q, \bar{Y}_Q, \bar{s}_{\mathtt{Com}}^{P}), \overline{\mathsf{tx}}^{\mathsf{s}})$ and update the channel space as $\Gamma^P(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathsf{c}}, r_P, R_Q, Y_Q, s_{\mathsf{c}}^{P}), \mathtt{TX}_{\mathsf{s}})$, for $\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^{P}, s_{\mathsf{s}}^{Q}\})$ and $\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^{Q}, y_P)\})$., and send $(\text{revokeP}, \mathsf{id}, \bar{r}_P) \xrightarrow{t_1+2} Q$. Else, execute $\mathtt{L\text{–}ForceClose}^P(\mathsf{id})$ and stop.

Party Q

11. Parse $\Gamma^Q(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\mathtt{TX}}_{\mathsf{c}}, \bar{r}_Q, \bar{R}_P, \bar{Y}_P, \bar{s}_{\mathtt{Com}}^{Q}), \overline{\mathsf{tx}}^{\mathsf{s}})$. If $(\text{revokeP}, \mathsf{id}, \bar{r}_P) \xleftarrow{\tau_1+2} P$, s.t. $(\bar{R}_P, \bar{r}_P) \in R$, $(\mathtt{REVOKE\text{–}REQ}, \mathsf{id}) \xrightarrow{\tau_1+2} \mathcal{E}$. Else execute $\mathtt{L\text{–}ForceClose}^Q(\mathsf{id})$ and stop.
12. If $(\mathtt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_1+2} \mathcal{E}$ as a reply, set

$$\Theta^Q(\mathsf{id}) := \Theta^Q(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathsf{c}}], \bar{r}_P, \bar{Y}_P, \bar{s}_{\mathtt{Com}}^{Q})$$
$$\Gamma^Q(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathsf{c}}, r_Q, R_P, Y_P, s_{\mathsf{c}}^{Q}), \mathtt{TX}_{\mathsf{s}}),$$

for $\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^{P}, s_{\mathsf{s}}^{Q}\})$, $\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_Q}([\mathtt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^{P}, y_Q)\})$, and send $(\text{revokeQ}, \mathsf{id}, \bar{r}_Q) \xrightarrow{\tau_1+2} P$. In the next round $(\mathtt{UPDATED}, \mathsf{id}) \xrightarrow{\tau_1+3} \mathcal{E}$ and stop. Else, in round $\tau_1 + 2$, execute $\mathtt{L\text{–}ForceClose}^Q(\mathsf{id})$ and stop.

Party P

13. If $(\text{revokeQ}, \mathsf{id}, \bar{r}_Q) \xleftarrow{t_1+4} Q$ s.t. $(\bar{R}_Q, \bar{r}_Q) \in R$, then set $\Theta^P(\mathsf{id}) := \Theta^P(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathsf{c}}], \bar{r}_Q, \bar{Y}_Q, \bar{s}_{\mathtt{Com}}^{P})$ and $(\mathtt{UPDATED}, \mathsf{id}) \xrightarrow{t_1+4} \mathcal{E}$. Else execute $\mathtt{L\text{–}ForceClose}^P(\mathsf{id})$ and stop.

Close

Party P upon $(\mathtt{CLOSE}, \mathsf{id}) \xleftarrow{t_0} \mathcal{E}$

1. Extract tx^{f} and $\mathtt{TX}_{\mathsf{s}}$ from $\Gamma^P(\mathsf{id})$ and set:

$$[\overline{\mathsf{tx}}^{\mathsf{s}}] := \mathtt{GenSplit}(\mathsf{tx}^{\mathsf{f}}.\mathsf{txid} \| 1, \mathtt{TX}_{\mathsf{s}}.\mathsf{output})$$

2. Compute $s_{\mathsf{s}}^{P} \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\overline{\mathsf{tx}}^{\mathsf{s}}])$ and send $(\text{peaceful–close}, \mathsf{id}, s_{\mathsf{s}}^{P}) \xrightarrow{t_0} Q$.

3. If $(\text{peaceful–close}, \mathsf{id}, s_{\mathsf{s}}^{Q}) \xleftrightarrow{t_0+1} Q$ s.t. $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\overline{\mathsf{tx}}^{\mathsf{s}}]; s_{\mathsf{s}}^{Q}) = 1$, set $\overline{\mathsf{tx}}^{\mathsf{s}} := ([\overline{\mathsf{tx}}^{\mathsf{s}}], \{s_{\mathsf{s}}^{P}, s_{\mathsf{s}}^{Q}\})$ and send $(\text{post}, \overline{\mathsf{tx}}^{\mathsf{s}}) \xleftrightarrow{t_0+1} \mathcal{L}$. Else, execute $\mathtt{L–ForceClose}^{P}(\mathsf{id})$ and stop.
4. Let $t_2 \leq t_1 + \Delta$ be the round in which $\overline{\mathsf{tx}}^{\mathsf{s}}$ is accepted by $\mathcal{L}$. Set $\Gamma^{P}(\mathsf{id}) := \bot$, $\Theta^{P}(\mathsf{id}) := \bot$ and send $(\mathtt{CLOSED}, \mathsf{id}) \xhookrightarrow{t_2} \mathcal{E}$.

Punish

Party P upon $\mathtt{PUNISH} \xhookleftarrow{t_0} \mathcal{E}$:

For each $\mathsf{id} \in \{0,1\}^*$ s.t. $\Theta^{P}(\mathsf{id}) \neq \bot$:

1. Parse $\Theta^{P}(\mathsf{id}) := \{([\mathtt{TX}_{\mathtt{c}}^{(\mathtt{i})}], r_Q^{(i)}, Y_Q^{(i)}, s^{(i)})\}_{i \in m}$ and extract γ from $\Gamma^{P}(\mathsf{id})$. If for some $i \in [m]$, there exist a transaction tx on $\mathcal{L}$ such that $\mathsf{tx.txid} = \mathtt{TX}_{\mathtt{c}}^{(\mathtt{i})}.\mathsf{txid}$, then parse the witness as $(s_P, s_Q) := \mathsf{tx.Witness})$, where $\mathsf{Vrfy}_{\mathsf{pk}_P}([\mathsf{tx}]; s_P) = 1$, and set $y_Q^{(i)} := \mathsf{Ext}(s_P, s^{(i)}, Y_Q^{(i)})$.
2. Define the body of the punishment transaction $[\mathtt{TX}_{\mathtt{pun}}]$ as:

$$\mathtt{TX}_{\mathtt{pun}}.\mathsf{input} := \mathsf{tx.txid} \| 1,$$
$$\mathtt{TX}_{\mathtt{pun}}.\mathsf{output} := \{(\gamma.\mathsf{cash}, \mathtt{One–Sig}_{\mathsf{pk}_P})\}$$

3. Sign $s_y \leftarrow \mathsf{Sign}_{y_Q^{(i)}}([\mathtt{TX}_{\mathtt{pun}}])$, $s_r \leftarrow \mathsf{Sign}_{r_Q^{(i)}}([\mathtt{TX}_{\mathtt{pun}}])$, $s_P \leftarrow \mathsf{Sign}_{\mathsf{pk}_P}([\mathtt{TX}_{\mathtt{pun}}])$, and set $\mathtt{TX}_{\mathtt{pun}} := ([\mathtt{TX}_{\mathtt{pun}}], s_y, s_r, s_P)$. Then $(\text{post}, \mathtt{TX}_{\mathtt{pun}}) \xhookrightarrow{t_0} \mathcal{L}$.
4. Let $\mathtt{TX}_{\mathtt{pun}}$ be accepted by $\mathcal{L}$ in round $t_1 \leq t_0 + \Delta$. Set $\Theta^{P}(\mathsf{id}) := \bot$, $\Gamma^{P}(\mathsf{id}) := \bot$ and output $(\mathtt{PUNISHED}, \mathsf{id}) \xhookrightarrow{t_1} \mathcal{E}$.

Subprocedures

$\mathtt{GenFund}(\mathsf{tid}, \gamma)$:
Return $[\mathsf{tx}]$, where $\mathsf{tx.input} := \mathsf{tid}$ and $\mathsf{tx.output} := \{(\gamma.\mathsf{cash}, \mathtt{Multi–Sig}_{\gamma.\mathsf{users}})\}$.

$\mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], (\mathsf{pk}_P, R_P, Y_P), (\mathsf{pk}_Q, R_Q, Y_Q))$:
Let $(c, \mathtt{Multi–Sig}_{\mathsf{pk}_P, \mathsf{pk}_Q}) := \mathsf{tx}^{\mathsf{f}}.\mathsf{output}[1]$ and denote

$$\varphi_1 := \mathtt{Multi–Sig}_{\mathsf{ToKey}(R_Q), \mathsf{ToKey}(Y_Q), \mathsf{pk}_P},$$
$$\varphi_2 := \mathtt{Multi–Sig}_{\mathsf{ToKey}(R_P), \mathsf{ToKey}(Y_P), \mathsf{pk}_Q},$$
$$\varphi_3 := \mathtt{CheckRelative}_{\Delta} \wedge \mathtt{Multi–Sig}_{\mathsf{pk}_P, \mathsf{pk}_Q}.$$

Return $[\mathsf{tx}]$, where $\mathsf{tx.input} = \mathsf{tx}^{\mathsf{f}}.\mathsf{txid} \| 1$ and $\mathsf{tx.output} := (c, \varphi_1 \vee \varphi_2 \vee \varphi_3)$.

$\texttt{GenSplit}(\mathsf{tid}, \theta)$:
Return [tx], where $\mathsf{tx.input} := \mathsf{tid}$ and $\mathsf{tx.output} := \theta$.

$\texttt{L–ForceClose}^P(\mathsf{id})$:
Let t_0 be the current round.

1. Extract $\texttt{TX}_\mathsf{c}$ and $\texttt{TX}_\mathsf{s}$ from $\Gamma(\mathsf{id})$ and send $(\text{post}, \texttt{TX}_\mathsf{c}) \overset{t_0}{\hookrightarrow} \mathcal{L}$.
2. Let $t_1 \leq t_0 + \Delta$ be the round in which $\texttt{TX}_\mathsf{c}$ is accepted by the blockchain. Wait for Δ rounds to $(\text{post}, \texttt{TX}_\mathsf{s}) \overset{t_1+\Delta}{\hookrightarrow} \mathcal{L}$.
3. Once $\texttt{TX}_\mathsf{s}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + 2\Delta$, set $\Theta^P(\mathsf{id}) := \bot$ and $\Gamma^P(\mathsf{id}) := \bot$ and output $(\texttt{CLOSED}, \mathsf{id}) \overset{t_2}{\hookrightarrow} \mathcal{E}$.

Remark 2. *In the protocol described in this section, we assume statement/witness pairs of R are valid key pairs. This assumption can be eliminated by modifying our protocol as follows. When creating a new commit transaction, each party samples the publishing pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and chooses a random revocation secret r_P. Thereafter, it computes a hash of both secrets as $h_P := \mathcal{H}(r_P)$ and $H_P := \mathcal{H}(y_P)$ and sends Y_P* ***and the hash values*** *h_P, H_P to the other party. In addition, it proves in zero knowledge the consistency of Y_P and H_P. The punishment mechanism for party P in the commit transaction then expects (i) a preimage of h_P (ii) a preimage of H_P and (iii) valid signature w.r.t. pk_Q.*

B.6 Simplifying functionality description

The formal description of the functionality $\mathcal{F}_L(T_p, k)$ as presented in Figure 3.4 is simplified. Namely, several natural checks that one would expect an ideal functionality to make when receiving a message are excluded from its description. For example, a functionality should ignore a message that is malformed (e.g. missing or additional parameters), requests an update of a channel that was never created, etc. We now define all these checks using a wrapper $\mathcal{W}_{\text{checks}}(T_p, k)$. Before we present the wrapper formally, let us discuss it at a high level.

Channel creation. Upon receiving a $(\texttt{CREATE}, \gamma, \mathsf{tid})$ message from a party P, the wrapper verifies that γ is a valid generalized channel, that its identifier is unique and that P is indeed a channel user. Moreover, the wrapper checks that the initial state of the channel has only two outputs – each spendable by one of the channel users only. Let us stress that while we do not support creation of a channel that already funds some off-chain applications, the application of interest can be added immediately after the channel creation is completed via a channel update. Finally, the wrapper verifies that tid refers to an output that is spendable by P and contains a sufficient amount of coins.

Channel update. Upon receiving a $(\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}})$ message from a party P, the wrapper verifies that the channel with identifier id exists and that P is a user of this channel. Moreover, the wrapper verifies the validity of the new state. This means that the outputs contained in the state are not distributing more coins than what is locked

in the channel and the conditions of the outputs are valid scripts of the underlying ledger. Finally, the wrapper verifies that there is no parallel update of the channel being performed and the channel is not being closed. Let us stress that this does not imply that applications built on top of the channel cannot be executed in parallel. This only says that all applications built on top of the channel must be created and closed at the same time.

Channel closure. We do not allow closing requests during a channel update or when a closure it already happening. Otherwise, the checks performed upon receiving a $(\texttt{CLOSE}, \mathsf{id})$ message from a party P are rather straightforward. The wrapper verifies that the channel with identifier id exists and that P is a user of that channel.

Functionality wrapper: $\mathcal{W}_{\text{checks}}(T_p, k)$

Below, we abbreviate $\mathcal{F} := \mathcal{F}_L(T_p, k)$.

Create: Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}) \overset{\tau_0}{\hookleftarrow} P$, where $P \in \gamma.\mathsf{users}$, check if: $\Gamma(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created; γ is valid according to the definition given in Section 3.4; $\gamma.\mathsf{st} = \{(c_P, \texttt{One-Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One-Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$; and there exists $(t, \mathsf{id}, i, \theta) \in \mathcal{L}.\texttt{UTXO}$ such that $\theta = (c_P, \texttt{One-Sig}_P)$ for $(\mathsf{id}, i) := \mathsf{tid}$;[a] If one of the above checks fails, drop the message. Else proceed as $\mathcal{F}$.

Update: Upon $(\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$ check if: $\gamma := \Gamma(\mathsf{id}) \neq \bot$; $P \in \gamma.\mathsf{users}$; there is no other update being preformed and the channel is not being closed; let $\theta = (\theta_1, \ldots \theta_\ell) = ((c_1, \varphi_1), \ldots, (c_\ell, \varphi_\ell))$, then $\sum_{j \in [\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $(\texttt{SETUP-OK}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$ check if: the message is a reply to the message $(\texttt{SETUP}, \mathsf{id}, \mathsf{tid})$ sent to P in round τ_0' such that $\tau_0 - \tau_0' \leq t_{\mathsf{stp}}$[b]. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $(\texttt{UPDATE-OK}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to the message $(\texttt{SETUP-OK}, \mathsf{id})$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $(\texttt{REVOKE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to either the message $(\texttt{UPDATE-OK}, \mathsf{id})$ sent to P in round τ_0 or the message $(\texttt{REVOKE-REQ}, \mathsf{id})$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.

Close: Upon $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if $\gamma := \Gamma(\mathsf{id}) \neq \bot$ and $P \in \gamma.\mathsf{users}$ and γ is currently not being updated or closed. If not, drop the message. Else proceed as $\mathcal{F}$.

All other messages are dropped.

[a] In case more channels are being created at the same time, then none of the other creation requests can use of the tid.

[b] What we formally mean by "reply" is explained in Appendix B.1.

B.7 Simplifying the protocol descriptions

Similarly to the descriptions of our ideal functionality, the description of the protocol Π_L presented in Appendix B.5 excludes many natural checks that an honest party should make in order to realize the ideal functionality. We define all these checks as a wrapper $\mathcal{W}_{\text{checksP}}$ which we first discuss at a high level and only then present formally.

Channel creation. When an honest party receives the message $(\texttt{CREATE}, \gamma, \mathsf{tid}_P)$ from the environment, she verifies that she is a user of the channel and that the channel is correctly formed. Moreover, she verifies that the channel identifier is unique. Finally, she checks that the transaction identifier tid_P refers to a published output that has the right amount of coins and belongs to her. If all the checks pass, party P behaves as described in the simplified protocol.

Similarly, when P receives the transaction identifier tid_Q from the other channel users, she first verifies that tid_Q refers to an output controlled by Q. Let us stress that skipping this check would be very dangerous for P. Malicious party Q could try to trick honest P to fund the channel completely on her own by proposing tid_Q that refers to an output controlled by P. As P sings the initial commit transaction, she would give her consent to spend both tid_P and tid_Q.

Channel update. When an honest party receives the message $(\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}})$ from the environment, i.e., P is the initiating party of the update, she verifies that the channel exists in her channel space, that there is no other update being performed already and that the channel is not being closed. Moreover, she verifies that the new state is valid. This means that it is not distributing more coins than is locked in the channel and all the output conditions are supported by the underlying blockchain. If all checks pass, party P behaves as described in the simplified protocol.

Analogously, if party P receives the message $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_Q, Y_Q)$ from some party Q, she verifies that id refers to an existing channel between P and Q that is currently not being updated. Moreover, P verifies that the proposed new state $\overrightarrow{\theta}$ is valid. Thereafter, she proceeds as in the simplified protocol.

Channel closure. Upon receiving the message $(\texttt{CLOSE}, \mathsf{id})$ from the environment, party P verifies that there exists a channel with identifier id in her channel space. Moreover, it checks that there is no update currently being performed and that the channel is not being closed already.

Protocol wrapper: $\mathcal{W}_{\text{checksP}}$

Party $P \in \mathcal{P}$ proceeds as follows:

Create: Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ check if: $P \in \gamma.\mathsf{users}$; $\Gamma^P(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created; γ is valid according to the definition given in Section 3.4;

$\gamma.\mathsf{st} = \{(c_P, \texttt{One-Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One-Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$; there exists $(t, \mathsf{id}, i, \theta) \in \mathcal{L}.\texttt{UTXO}$ such that $\theta = (c_P, \texttt{One-Sig}_P)$ for $(\mathsf{id}, i) := \mathsf{tid}$. If one of the above checks fails, drop the message. Else proceed as in Π_L.

Upon $(\text{createInfo}, \mathsf{id}, \mathsf{tid}_Q, R_Q, Y_Q) \xleftarrow{\tau_0+1} Q$, check if: you accepted a $(\texttt{CREATE}, \gamma, \mathsf{tid})$ message in round τ_0 with $\gamma.\mathsf{id} = \mathsf{id}$; there exists $(t, \mathsf{id}, i, \theta) \in \mathcal{L}.\texttt{UTXO}$ such that $\theta = (c_Q, \texttt{One-Sig}_{pk_Q})$ for $(\mathsf{id}, i) := \mathsf{tid}_Q$ and $(c_Q, \texttt{One-Sig}_{\mathsf{pk}_Q}) \in \gamma.\mathsf{st}$; there is no other channel are being created using this tid_Q. If one of the above checks fails, drop the message. Else proceed as Π_L.

Update: Upon $(\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}}) \xleftarrow{\tau_0} \mathcal{E}$ check if: $\gamma := \Gamma^P(\mathsf{id}) \neq \bot$; there is no other update being preformed and the channel is not being closed; let $\theta = (\theta_1, \ldots \theta_\ell) = ((c_1, \varphi_1), \ldots, (c_\ell, \varphi_\ell))$, then $\sum_{j\in[\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If on of the checks fails, drop the message. Else proceed as in Π_L.

Upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_Q, Y_Q) \xleftarrow{\tau_0} Q$, check if $\{P, Q\} = \gamma.\mathsf{users}$; $\gamma := \Gamma(\mathsf{id}) \neq \bot$; there is no other update being preformed and the channel is not being closed; let $\theta = (\theta_1, \ldots \theta_\ell) = ((c_1, \varphi_1), \ldots, (c_\ell, \varphi_\ell))$, then $\sum_{j\in[\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If one of the above checks fails, drop the message. Else proceed as in Π_L.

Upon $(\texttt{SETUP-OK}, \mathsf{id}) \xleftarrow{\tau_0} \mathcal{E}$ check if: the message is a reply to the message $(\texttt{SETUP}, \mathsf{id}, \mathsf{tid})$ you sent in round τ_0' such that $\tau_0 - \tau_0' \leq t_{\mathsf{stp}}$[a]. If not, drop the message. Else proceed as in Π_L.

Upon $(\texttt{UPDATE-OK}, \mathsf{id}) \xleftarrow{\tau_0} \mathcal{E}$, check if the message is a reply to the message $(\texttt{SETUP-OK}, \mathsf{id})$ you sent in round τ_0. If not, drop the message. Else proceed as in Π_L.

Upon $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_0} \mathcal{E}$, check if the message is a reply to either $(\texttt{UPDATE-OK}, \mathsf{id})$ or $(\texttt{REVOKE-REQ}, \mathsf{id})$ you sent in round τ_0. If not, drop the message. Else proceed as in Π_L.

Close: Upon $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_0} \mathcal{E}$, check if $\gamma := \Gamma^P(\mathsf{id}) \neq \bot$ and that the channel is not being updated or closed. If one of the checks fails, drop the message. Else proceed as in Π_L.

Upon $(\text{peaceful-close}, \mathsf{id}, \sigma_Q) \xleftarrow{\tau_1} Q$, check if you sent $(\text{peaceful-close}, \mathsf{id}, \sigma_P) \xrightarrow{\tau_1-1} Q$. If not, then drop the message. Else proceed as in Π_L.

All other messages are dropped.

[a] What we formally mean by "reply" is explained in Appendix B.1.

B.8 Security proof

In this section, we provide a proof for Theorem 3. In our proof, we provide the code for a simulator, that simulates the protocol $\Pi_L^{\mathcal{L}(\Delta,\Sigma)}(\Xi_{R,\Sigma})$ in the ideal world having access to the functionalities $\mathcal{L}$ and $\mathcal{F}_L$. The main challenge in providing a simulation in UC proofs usually arises from the fact that the simulator is not given the secret inputs of the parties in the protocol, which makes it difficult to provide a simulated transcript that is indistinguishable from a transcript of a real protocol execution. However, in our setting, parties do not obtain any secret inputs, but only receive commands from the environment $\mathcal{E}$, and hence the only challenge that arises during the simulation is handling different behavior of malicious parties. For this reason, we omit the simulation for the case where both parties are honest in the protocol. Furthermore, due to the same reason,

as long as the protocol can be simulated in the ideal world, the ideal and real-world executions are indistinguishable. We emphasize that the security of the protocol and its realizability rely on the correctness and security properties of the underlying adaptor signature scheme, namely unforgeability, witness intractability, and adaptability.

Let us now explain the necessity of the adaptor signature properties in more detail. Clearly, if the environment or malicious parties are able to generate signatures on behalf of honest parties, we create an adversary that can use them in order to win the unforgeability game of the adaptor signature scheme. Therefore, only the simulator can generate valid signatures on behalf of the honest parties (the environment can do so only upon guessing the correct signing keys, which happens only with negligible probability). Witness Extractability is necessary in order to punish the dishonest party who has published an old commit transaction. Hence, if a malicious party can publish a valid signature for which the extract algorithm Ext, in step 1 of the simulation for the punish procedure, does not output a correct witness, we can build an adversary that can win the witness extractability game of the adaptor signature scheme. Further, adaptability is required in order to complete the pre-signature of the new commit transaction. Therefore, if a malicious party can generate a pre-signature that cannot be adapted, in step 8 of the simulation for the update procedure, we can build an adversary who can break the pre-signature adaptability property. Last but not least, the signatures generated upon adapting a pre-signature are valid according to correctness and hence the punish transaction generated in step 3 of the simulation for the punish procedure, is signed correctly and will get accepted by the blockchain.

Remark 3. *In the following proof, we use the witness extracted from an adaptor signature as a signing secret key. We note that the proof extends naturally to the case where the witness is used as a hash preimage even though this requires an additional zero-knowledge proof, which guarantees consistency of the hash value and the preimage.*

Simulator for creating generalized channels

Let $T_1 = 3$.

Case A is honest and B is corrupted

Upon A sending $(\texttt{CREATE}, \gamma, \mathsf{tid}_A) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$, if B does not send $(\texttt{CREATE}, \gamma, \mathsf{tid}_B) \overset{\tau}{\hookrightarrow} \mathcal{F}_L$ where $|\tau_0 - \tau| \leq T_1$, then distinguish the following cases:

1. If B sends $(\text{createInfo}, \mathsf{id}, \mathsf{tid}_B, R_B, Y_B) \overset{\tau_0}{\hookrightarrow} A$, then send $(\texttt{CREATE}, \gamma, \mathsf{tid}_B) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ on behalf of B.
2. Otherwise stop.

Do the following:

1. Set $\mathsf{id} := \gamma.\mathsf{id}$, generate a revocation public/secret pair $(R_A,\ r_A) \leftarrow \mathsf{GenR}(pp)$, generate publishing public/secret pair $(Y_A, y_A) \leftarrow \mathsf{GenR}(pp)$ and send $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_A, R_A, Y_A) \overset{\tau_0}{\hookrightarrow} B$.
2. If you receive $(\mathsf{createInfo}, \mathsf{id}, \mathsf{tid}_B, R_B, Y_B) \overset{\tau_0+1}{\hookleftarrow} B$, create the body of the funding, the first commit and split transactions:

$$[\mathsf{tx}^{\mathsf{f}}] := \mathtt{GenFund}((\mathsf{tid}_A, \mathsf{tid}_B), \gamma)$$
$$[\mathtt{TX}_{\mathsf{c}}] := \mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B, 0)$$
$$[\mathtt{TX}_{\mathsf{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \gamma.\mathsf{st})$$

   where $I_A := (\mathsf{pk}_A, R_A, Y_A)$ and $I_B := (\mathsf{pk}_B, R_B, Y_B)$. Else stop.
3. Pre-sign $[\mathtt{TX}_{\mathsf{c}}]$ w.r.t. Y_B and sign $[\mathtt{TX}_{\mathsf{s}}]$,

$$s_{\mathsf{c}}^A \leftarrow \mathsf{pSign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}], Y_B)$$
$$s_{\mathsf{s}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{s}}])$$

   and $(\mathsf{createCom}, \mathsf{id}, s_{\mathsf{c}}^A, s_{\mathsf{s}}^A) \overset{\tau_0+1}{\hookrightarrow} B$.
4. If you receive $(\mathsf{createCom}, \mathsf{id}, s_{\mathsf{c}}^B, s_{\mathsf{s}}^B) \overset{\tau_0+2}{\hookleftarrow} B$, s.t.

$$\mathsf{pVrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{c}}], Y_A; s_{\mathsf{c}}^B) = 1$$
$$\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^B) = 1$$

   sign the funding transaction $s_{\mathsf{f}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\mathsf{tx}^{\mathsf{f}}])$ and $(\mathsf{createFund}, \mathsf{id}, s_{\mathsf{f}}^A) \overset{\tau_0+2}{\hookrightarrow} B$. Else stop.
5. If you $(\mathsf{createFund}, \mathsf{id}, s_{\mathsf{f}}^B) \overset{\tau_0+3}{\hookleftarrow} B$ s.t. $\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathsf{tx}^{\mathsf{f}}]; s_{\mathsf{f}}^B) = 1$, define $\mathsf{tx}^{\mathsf{f}} := ([\mathsf{tx}^{\mathsf{f}}], \{s_{\mathsf{f}}^A, s_{\mathsf{f}}^B\})$ and $(\mathsf{post},\ \mathsf{tx}^{\mathsf{f}}) \overset{\tau_0+3}{\hookrightarrow} \mathcal{L}$. Else parse $(\theta_A,\ \theta_B) := \gamma.\mathsf{st}$, create tx such that $\mathsf{tx}.\mathsf{input} := \mathsf{tid}_A$, $\mathsf{tx}.\mathsf{output} := \theta_A$, $\mathsf{tx}.w \leftarrow \mathsf{Sign}_{\mathsf{pk}_A}([\mathsf{tx}])$ and $(\mathsf{post}, \mathsf{tx}) \overset{\tau_0+3}{\hookrightarrow} \mathcal{L}$ and stop.
6. If tx^{f} is accepted by $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + 3 + \Delta$, add

$$\Gamma^A(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathsf{c}}, r_A, R_B, Y_B, s_{\mathsf{c}}^A), \mathtt{TX}_{\mathsf{s}}),$$

   where $\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^A, s_{\mathsf{s}}^B\})$ and

$$\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^B, y_A)\}).$$

Simulator for updating generalized channels

Let $T_1 = 2$ and $T_2 = 1$ and let $|\mathsf{tid}| = 1$.

Case A is honest and B is corrupted

Upon A sending $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0} \mathcal{F}_L$, proceed as follows:

1. Generate new revocation public/secret pair $(R_P, \ r_P) \leftarrow \mathsf{GenR}$ and a new publishing public/secret pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_A, Y_A) \xhookrightarrow{\tau_0^A} B$.

2. Upon $(\text{updateInfo}, \mathsf{id}, h_B, Y_B, s_{\mathsf{s}}^B) \xhookleftarrow{\tau_0^A+2} B$, extract tx^{f} from $\Gamma^B(\mathsf{id})$ and

$$[\mathtt{TX}_{\mathsf{c}}] := \mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B)$$
$$[\mathtt{TX}_{\mathsf{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta),$$

for $I_A := (\mathsf{pk}_A, R_A, Y_A)$ and $I_B := (\mathsf{pk}_B, R_B, Y_B)$. If $\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^B) = 1$, send $(\texttt{SETUP}, \mathsf{id}, \mathtt{TX}_{\mathsf{s}}.\mathsf{txid}) \xhookrightarrow{\tau_0^A+2} \mathcal{E}$. Else stop.

3. If A sends $(\texttt{SETUP–OK}, \ \mathsf{id}) \xhookrightarrow{\tau_1^A \leq \tau_0^A+2+t_{\mathsf{stp}}} \mathcal{F}_L$, compute $s_{\mathsf{c}}^A \leftarrow \mathsf{pSign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}], \ Y_B)$ and $s_{\mathsf{s}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{s}}])$, and send $(\text{update–commitA}, \mathsf{id}, s_{\mathsf{c}}^A, s_{\mathsf{s}}^A) \xhookrightarrow{\tau_1^A} B$.

4. In round $\tau_1^A + 2$ distinguish the following cases:

   - If you receive $(\text{update–commitB}, \ \mathsf{id}, \ s_{\mathsf{c}}^B) \xhookleftarrow{\tau_1^A+2} B$ and if B has not sent $(\texttt{UPDATE–OK}, \ \mathsf{id}) \xhookrightarrow{\tau_1^A+1} \mathcal{F}_L$, then send $(\texttt{UPDATE–OK}, \ \mathsf{id}) \xhookrightarrow{\tau_1^A+1} \mathcal{F}_L$ on behalf of B. If $\mathsf{pVrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{c}}], Y_A; s_{\mathsf{c}}^B) = 0$, then stop.
   - If you receive $(\text{updateNotOk}, \ \mathsf{id}, \ r_B) \xhookleftarrow{\tau_2^P+2} B$, where $(R_B, \ r_B) \in R$, add $\Theta^A(\mathsf{id}) := \Theta^A(\mathsf{id}) \cup ([\mathtt{TX}_{\mathsf{c}}], r_B, Y_B, s_{\mathsf{c}}^A)$, instruct $\mathcal{F}_L$ to stop and stop.
   - Else, execute the simulator code for the procedure $\mathtt{L–ForceClose}^A(\mathsf{id})$ and stop.

5. If A sends $(\texttt{REVOKE}, \ \mathsf{id}) \xhookrightarrow{\tau_1^A+2} \mathcal{F}_L$, then parse $\Gamma^A(\mathsf{id})$ as $(\gamma, \ \mathsf{tx}^{\mathsf{f}}, \ (\overline{\mathtt{TX}}_{\mathsf{c}}, \ \bar{r}_A, \ \bar{R}_B, \ \bar{Y}_B, \ \bar{s}_{\mathtt{Com}}^A), \ \overline{\mathsf{tx}}^{\mathsf{s}})$ and update the channel space as $\Gamma^A(\mathsf{id}) := (\gamma, \ \mathsf{tx}^{\mathsf{f}}, \ (\mathtt{TX}_{\mathsf{c}}, \ r_A, \ R_B, \ Y_B, \ s_{\mathsf{c}}^A)$, $\mathtt{TX}_{\mathsf{s}})$, for $\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \ \{s_{\mathsf{s}}^A, \ s_{\mathsf{s}}^B\})$ and $\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \ \{\mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}]), \ \mathsf{Adapt}(s_{\mathsf{c}}^B, \ y_A)\})$. Then send $(\text{revokeP}, \mathsf{id}, \bar{r}_A) \xhookrightarrow{\tau_1^A+2} B$. Else, execute the simulator code for the procedure $\mathtt{L–ForceClose}^A(\mathsf{id})$ and stop.

6. If you receive $(\text{revokeB}, \mathsf{id}, \bar{r}_B) \xhookleftarrow{\tau_1^A+4} B$ and if B has not sent $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^B+2} \mathcal{F}_L$, then send $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^B+2} \mathcal{F}_L$ on behalf of B. Check if $(\bar{R}_B, \bar{r}_B) \in R$, then set

$$\Theta^B(\mathsf{id}) := \Theta^A(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathsf{c}}], \bar{r}_B, \ \bar{Y}_B, \bar{s}_{\mathtt{Com}}^A)$$

Else execute the simulator code for the procedure $\mathtt{L–ForceClose}^A(\mathsf{id})$ and stop.

Case B is honest and A is corrupted

Upon A sending $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, h_A) \xhookrightarrow{\tau_0} B$, send $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0} \mathcal{F}_L$ on behalf of A, if A has not already sent this message. Proceed as follows:

1. Upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_A, Y_A) \xhookleftarrow{\tau_0^B} A$, generate $(R_B, r_B) \leftarrow \mathsf{GenR}$ and $(Y_B, y_B) \leftarrow \mathsf{GenR}$.
2. Extract tx^{f} from $\Gamma^A(\mathsf{id})$ and

$$[\texttt{TX}_{\mathsf{c}}] := \texttt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B)$$
$$[\texttt{TX}_{\mathsf{s}}] := \texttt{GenSplit}([\texttt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta)$$

   where $I_A := (\mathsf{pk}_A, R_A, Y_A)$, $I_B := (\mathsf{pk}_B, R_B, Y_B)$.
3. Compute $s_{\mathsf{s}}^B \leftarrow \mathsf{Sign}_{\mathsf{sk}_B}([\texttt{TX}_{\mathsf{s}}])$, send $(\text{updateInfo}, \mathsf{id}, R_B, Y_B, s_{\mathsf{s}}^B) \xhookrightarrow{\tau_0^B} A$.
4. If you $(\text{updateComP}, \mathsf{id}, s_{\mathsf{c}}^A, s_{\mathsf{s}}^A) \xhookleftarrow{\tau_1^B \leq \tau_0^B + 2 + t_{\mathsf{stp}}} A$ then send $(\texttt{SETUP\_OK}, \mathsf{id}) \xhookrightarrow{\tau_1^B} \mathcal{F}_L$ on behalf of A, if A has not sent this message.
5. Check if $\mathsf{pVrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{c}}], Y_Q; s_{\mathsf{c}}^P) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^P) = 1$.
6. If B sends $(\texttt{UPDATE\_OK}, \mathsf{id}) \xhookrightarrow{\tau_1^B} \mathcal{F}_L$, pre-sign $s_{\mathsf{c}}^B \leftarrow \mathsf{pSign}([\texttt{TX}_{\mathsf{c}}], Y_A)$ and send $(\text{updateComQ}, \mathsf{id}, s_{\mathsf{c}}^B) \xhookrightarrow{\tau_1^B} A$. Else send $(\text{updateNotOk}, \mathsf{id}, r_B) \xhookrightarrow{\tau_1^B} A$ and stop.
7. Parse $\Gamma^B(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\texttt{TX}_{\mathsf{c}}}, \bar{r}_B, \bar{R}_A, \bar{Y}_A, \bar{s}_{\texttt{Com}}^B), \overline{\mathsf{tx}^{\mathsf{s}}})$. If you $(\text{revokeP}, \mathsf{id}, \bar{r}_A) \xhookleftarrow{\tau_1^B + 2} A$, send $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^B + 2} \mathcal{F}_L$ on behalf of A, if A has not sent this message.

   Else if you do not receive $(\text{revokeP}, \mathsf{id}, \bar{r}_A) \xhookleftarrow{\tau_1^B + 2} A$ or if $(\bar{R}_A, \bar{r}_A) \notin R$, execute the simulator code of the procedure $\texttt{L\_ForceClose}^B(\mathsf{id})$ and stop.
8. If B sends $(\texttt{REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^B + 2} \mathcal{F}_L$, then set

$$\Theta^B(\mathsf{id}) := \Theta^B(\mathsf{id}) \cup ([\overline{\texttt{TX}_{\mathsf{c}}}], \bar{r}_A, \bar{Y}_A, \bar{s}_{\texttt{Com}}^B)$$
$$\Gamma^B(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\texttt{TX}_{\mathsf{c}}, r_B, R_A, Y_A, s_{\mathsf{c}}^B), \texttt{TX}_{\mathsf{s}}),$$

   for $\texttt{TX}_{\mathsf{s}} := ([\texttt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^A, s_{\mathsf{s}}^B\})$ and $\texttt{TX}_{\mathsf{c}} := ([\texttt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_B}([\texttt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^A, y_B)\})$. Then $(\text{revokeB}, \mathsf{id}, \bar{r}_B) \xhookrightarrow{\tau_1^B + 2} A$ and stop. Else, in round $\tau_1^B + 2$, execute the simulator code of the procedure $\texttt{L\_ForceClose}^B(\mathsf{id})$ and stop.

Simulator for closing generalized channels

Let $T_1 = 1$.

Case A is honest and B is corrupted

Upon A sending $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$, if B does not send $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookrightarrow} \mathcal{F}_L$ where $|\tau_0 - \tau| \leq T_1$, then distinguish the following cases:

1. If B sends $s_{\mathsf{s}}^B \overset{\tau_0}{\hookrightarrow} A$, then send $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ on behalf of B.
2. Otherwise execute the simulator code of the procedure $\texttt{L–ForceClose}^A(\mathsf{id})$ and stop.

1. Extract tx^{f} and $\texttt{TX}_{\mathsf{s}}$ from $\Gamma^A(\mathsf{id})$. Create the body of the final split transaction $[\overline{\mathsf{tx}}^{\mathsf{s}}]$ as follows
$$[\overline{\mathsf{tx}}^{\mathsf{s}}] := \texttt{GenSplit}(\mathsf{tx}^{\mathsf{f}}.\mathsf{txid} \| 1, \texttt{TX}_{\mathsf{s}}.\mathsf{output})$$
2. Compute the signature $s_{\mathsf{s}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\overline{\mathsf{tx}}^{\mathsf{s}}])$ and send $s_{\mathsf{s}}^A \overset{\tau_0}{\hookrightarrow} B$.
3. If you receive $s_{\mathsf{s}}^B \overset{\tau_0+1}{\hookleftarrow} B$, s.t. $\mathsf{Vrfy}_{\mathsf{pk}_B}([\overline{\mathsf{tx}}^{\mathsf{s}}]; s_{\mathsf{s}}^B) = 1$, set $\overline{\mathsf{tx}}^{\mathsf{s}} := ([\overline{\mathsf{tx}}^{\mathsf{s}}], \{s_{\mathsf{s}}^A, s_{\mathsf{s}}^B\})$ and send $(\mathsf{post}, \overline{\mathsf{tx}}^{\mathsf{s}}) \overset{\tau_0+1}{\hookrightarrow} \mathcal{L}$. Else, execute the simulator code for the procedure $\texttt{L–ForceClose}^A(\mathsf{id})$ and stop.
4. Let $\tau_2 \leq \tau_1 + \Delta$ be the round in which $\overline{\mathsf{tx}}^{\mathsf{s}}$ is accepted by the blockchain. Set $\Gamma^A(\mathsf{id}) = \bot$, $\Theta^A(\mathsf{id}) = \bot$.

Simulator for punishment of generalized channels

Case A is honest and B is corrupted

Upon A sending $\texttt{PUNISH} \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$, for each $\mathsf{id} \in \{0,1\}^*$ such that $\Theta^P(\mathsf{id}) \neq \bot$ do the following:

1. Parse $\Theta^A(\mathsf{id}) := \{([\texttt{TX}_{\mathsf{c}}^{(\mathsf{i})}], r_B^{(i)}, Y_A^{(i)}, s^{(i)})\}_{i \in m}$ and extract γ from $\Gamma^A(\mathsf{id})$. If for some $i \in [m]$, there exist a transaction tx on $\mathcal{L}$ such that $\mathsf{tx}.\mathsf{txid} = \texttt{TX}_{\mathsf{c}}^{(\mathsf{i})}.\mathsf{txid}$, then parse the witness as $(s_A, s_B) := \mathsf{tx}.\mathsf{Witness})$, where $\mathsf{Vrfy}_{\mathsf{pk}_A}([\mathsf{tx}]; s_A) = 1$, and set $y_B^{(i)} := \mathsf{Ext}(s_A, s^{(i)}, Y_B^{(i)})$.
2. Define the body of the punishment transaction $[\texttt{TX}_{\mathsf{pun}}]$ as:
$$\texttt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tx}.\mathsf{txid} \| 1,$$
$$\texttt{TX}_{\mathsf{pun}}.\mathsf{output} := \{(\gamma.\mathsf{cash}, \texttt{One–Sig}_{\mathsf{pk}_A})\}$$
3. Compute the signatures $s_y \leftarrow \mathsf{Sign}_{y_B^{(i)}}([\texttt{TX}_{\mathsf{pun}}]), s_r \leftarrow \mathsf{Sign}_{r_B^{(i)}}([\texttt{TX}_{\mathsf{pun}}]), s_A \leftarrow \mathsf{Sign}_{\mathsf{pk}_A}([\texttt{TX}_{\mathsf{pun}}])$, and set $\texttt{TX}_{\mathsf{pun}} := ([\texttt{TX}_{\mathsf{pun}}], s_y, s_r, s_A)$. Then $(\mathsf{post}, \texttt{TX}_{\mathsf{pun}}) \overset{\tau_0}{\hookrightarrow} \mathcal{L}$.
4. Let $\texttt{TX}_{\mathsf{pun}}$ be accepted by $\mathcal{L}$ in round $\tau_1 \leq \tau_0 + \Delta$. Set $\Theta^A(\mathsf{id}) = \bot$, $\Gamma^A(\mathsf{id}) = \bot$.

Simulator for $\texttt{ForceClose}^P(\mathsf{id})$

Let τ_0 be the current round

1. Extract $\texttt{TX}_{\mathsf{c}}$ and $\texttt{TX}_{\mathsf{s}}$ from $\Gamma(\mathsf{id})$.

2. Send $(\text{post}, \texttt{TX}_\texttt{c}) \overset{\tau_0}{\hookrightarrow} \mathcal{L}$.
3. Let $\tau_1 \leq \tau_0 + \Delta$ be the round in which $\texttt{TX}_\texttt{c}$ is accepted by the blockchain. Wait for Δ rounds to $(\text{post}, \texttt{TX}_\texttt{s}) \overset{\tau_2+\Delta}{\hookrightarrow} \mathcal{L}$.
4. Once $\texttt{TX}_\texttt{s}$ is accepted by the blockchain in round $\tau_3 \leq \tau_2 + 2\Delta$, set $\Theta^P(\mathsf{id}) = \bot$ and $\Gamma^P(\mathsf{id}) = \bot$.

B.9 Applications on top of generalized channels

We summarize the general discussion from Section 3.7 about which applications can be built on top of generalized channels in Remark 4, where we denote a two-party application π whose funding source can be published within t rounds as $\pi(t)$. Thus, $\pi(0)$ indicates that π is funded directly by the ledger.

Remark 4 (Lifting on-chain functionality off-chain)**.** *Let $\pi(0)$ be an application executed between two parties P_1 and P_2 funded directly by a ledger $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$, where $\mathcal{V}$ allows at least for transaction authorization w.r.t. Σ, relative time-locks and constant number of Boolean operations $\wedge$ and $\vee$. Then $\pi(3\Delta)$ can be funded by a generalized channel between P_1 and P_2, hence executed fully off-chain, while guaranteeing* instant finality with punish *to both parties. This means that either $\pi(3\Delta)$ terminates as $\pi(0)$ would over $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$, or the honest party is financially compensated.*

In Section 3.7, we described how to construct two concrete applications on top of generalized channels. The process described there can naturally be generalized to any two-party applications which is what we do in this section.

Assume that two parties already created a generalized channel γ via $\mathcal{F}_L$ and now want to use it for several applications. For that, parties have to carry out the following steps.

Initialize: Parties agree on the new state θ of γ and the upper bound t_{stp} on the time required to set up applications. That is, for each application parties agree on (i) the amount of coins they want to invest and the funding condition; technically, this means that parties define $\theta_i = (\theta_i.\mathsf{cash}, \theta_i.\varphi)$, and (ii) the maximal set up time t_i. The value t_{stp} is defined as $\max_i t_i$, thereby upper-bounding the number of rounds that it takes to set up all the applications in parallel.

Prepare: One party sends the message $(\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}})$ to $\mathcal{F}_L$ in order to prepare the update. Upon receiving such message, $\mathcal{F}_L$ responds with tid– a vector of k transaction identifiers referring to transactions that contain the output vector θ and hence are candidate funding sources of our applications.

Setup: For every $\mathsf{tid}_j \in \mathsf{tid}$, parties exchange the application-dependent information required to fulfill the conditions $\{\theta_i.\varphi\}$.

Complete: Parties inform $\mathcal{F}_L$ about setup completion by sending $\texttt{SETUP–OK}$ and $\texttt{UPDATE–OK}$ messages. Thereafter, $\mathcal{F}_L$ requests both parties to revoke the old state

of γ which they do by invoking $\mathcal{F}_L$ on input the message REVOKE. $\mathcal{F}_L$ notifies the users of the completed update via the message UPDATED.

B.9.1 Claim-Or-Refund

Initialize: First, parties need to agree on the new state of the channel that would include the new claim-or-refund applications. To this end, parties exchange the function f, decide on the time-out value t, and create three outputs (one for the conditional payment, one for the remaining balance of A and one for the balance B): (i) $\theta_0.\mathsf{cash} := \beta$, $\theta_0.\varphi := (\mathtt{Check}_f \wedge \mathtt{One\text{–}Sig}_{\mathsf{pk}_B}) \vee (\mathtt{CheckAbsolute}_t \wedge \mathtt{One\text{–}Sig}_{\mathsf{pk}_A})$; (ii) $\theta_1.\mathsf{cash} := \alpha_A - \beta$, $\theta_1.\varphi := \mathtt{One\text{–}Sig}_{\mathsf{pk}_A}$; and (iii) $\theta_2.\mathsf{cash} := \alpha_B$, $\theta_2.\varphi := \mathtt{One\text{–}Sig}_{\mathsf{pk}_B}$. The new channels state is then $\theta = (\theta_0, \theta_1, \theta_2)$.

Prepare: One party sends the message $(\mathtt{UPDATE}, \gamma.\mathsf{id}, \theta, 0)$ to $\mathcal{F}_L$ in order to prepare the update. The last coordinate is set to 0, because no special setup is needed in the case of the claim-or-refund application. Upon receiving such message, $\mathcal{F}_L$ responds with tid– a vector of k transaction identifiers referring to transactions that contain the output vector θ and hence are candidate funding sources of our applications.

Complete: Parties inform $\mathcal{F}_L$ about their intention to complete the update by sending SETUP–OK and UPDATE–OK messages. Thereafter, $\mathcal{F}_L$ requests both parties to revoke the old state of γ which they do by invoking $\mathcal{F}_L$ on input the message REVOKE. $\mathcal{F}_L$ notifies the users of the completed update via the message UPDATED.

A similar process is used when B wants to claim the β coins or A wants to refund β coins. Namely, if B wants to claim the β coins, this party initiates a new update of γ_0 s.t., $\alpha_A - \beta$ coins are assigned to A and $\alpha_B + \beta$ coins are assigned to B. The security of the solution follows from the fact that if the update fails, the channel is closed in the latest agreed state. Hence, the output funding the claim-or-refund is published in an on-chain transaction allowing B to claim the β coins over the blockchain. Analogously, for the refund of A.

B.9.2 Channel-Splitting

Initialize: Parties first agree on the new state of the channel. To this end, they create one output per sub-channel: (i) $\theta_0.\mathsf{cash} := \gamma_0.\mathsf{cash}$, $\theta_0.\varphi := \mathtt{One\text{–}Sig}_{\mathsf{pk}_A} \wedge \mathtt{One\text{–}Sig}_{\mathsf{pk}_B}$; and (ii) $\theta_1.\mathsf{cash} := \gamma_1.\mathsf{cash}$, $\theta_1.\varphi := \mathtt{One\text{–}Sig}_{\mathsf{pk}_A} \wedge \mathtt{One\text{–}Sig}_{\mathsf{pk}_B}$. The new state is hence $\theta = (\theta_0, \theta_1)$.

Prepare: As in the previous example, one party sends the message $(\mathtt{UPDATE}, \gamma.\mathsf{id}, \theta, 2)$ to $\mathcal{F}_L$ in order to prepare the update. This time, the setup time is set to 2 rounds as this is how long it takes to setup a new generalized channel. Upon receiving such message, $\mathcal{F}_L$ responds with tid– a vector of k transaction identifiers referring to transactions that contain the output vector θ and hence are candidate funding sources of our applications.

Setup: For each sub-channel, parties generate and sign the commit and split transactions representing the initial channel state. This procedure, explained in Section 3.6, takes 2 rounds.

Complete: Parties inform $\mathcal{F}_L$ about the completed setup by sending `SETUP–OK` and `UPDATE–OK` messages. Thereafter, $\mathcal{F}_L$ requests both parties to revoke the old state of γ which they do by invoking $\mathcal{F}_L$ on input the message `REVOKE`. $\mathcal{F}_L$ notifies the users of the completed update via the message `UPDATED`.

Remark 5. *The setup phase is run for each transaction identifier in* tid *which means that parties have to set up and maintain k copies of all their applications. Hence, low values of the parameter k are of great importance.*

APPENDIX C

Appendix to Chapter 4

C.1 Discussion on practical deployment

Payment fees. We encode a fee mechanism in our construction. For simplicity, we assume that every intermediary charges the same fee amount: fee. However, it is trivial to extend this mechanism to allow for different fees. The sender initially puts an amount $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$ in the output $\theta_{i,0}$. Every intermediary now deducts fee from this amount when opening the construction with its own right neighbor. Specifically, an intermediary U_i receives α_{i-1} and forwards only $\alpha_i := \alpha_{i-1} - \mathsf{fee}$. Thereby, every intermediary effectively gains fee coins in the case of a successful payment.

Refund tradeoff. In the case of a refund, where a fast refund (see Section 4.3) is not possible, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$. Doing this will have the cost of publishing this transaction (and possibly the transaction containing its input) plus the $(n-1) \cdot \epsilon$ that go to the intermediaries. The amount ϵ can be the smallest possible amount of cash since it is just used to enable the payment. In other words, for Bitcoin, we can say $\epsilon := 1$ satoshi,[1] which is currently around 0.00011 USD. However, the refund of Blitz payments has a fundamental advantage over the one in the Lightning Network (LN). The refund time is only constant in the worst case and if the sender is honest, is only the time it takes to publish $\mathsf{tx}^{\mathsf{er}}$ (i.e., Δ) instead of $n \cdot \xi$. We presented this advantage in Section 4.3.

So the tradeoff is a more expensive, but much faster refund. This immensely reduces the effect of griefing attacks and increases the overall transaction throughput.

[1] In practice, Bitcoin transactions need to carry a total amount of one dust, which is 546 satoshis. Having individual outputs of one satoshi is not a problem, as the sender can include an additional output to a stealth address under its control, such that the sum is greater than one dust. In $\mathsf{tx}^{\mathsf{r}}_i$, the output of $\mathsf{tx}^{\mathsf{er}}$ holding one satoshi is combined with the first output of the state $\mathsf{tx}^{\mathsf{state}}$, resulting in a sum larger than one dust.

Race. We already mentioned that only the sender can publish $\mathsf{tx}^{\mathsf{er}}$ and because of the time delays, the timing is the same for every user on the path. We claimed that the latest possible time to safely publish $\mathsf{tx}^{\mathsf{er}}$ and still be able to claim the refund is $T - t_{\mathsf{c}} - 3\Delta$. However, there is a time frame after $T - t_{\mathsf{c}} - 3\Delta$ up until $T - t_{\mathsf{c}} - 2\Delta$, where the sender could publish $\mathsf{tx}^{\mathsf{er}}$ and still, $\mathsf{tx}_i^{\mathsf{r}}$ would be sent to the ledger before time T. However now, everyone is at risk, because we said that accepting a transaction takes at most Δ time and at time T, already $\mathsf{tx}_i^{\mathsf{p}}$ might be sent to the ledger and there might be a race over which of these two transactions is accepted first. We argue, that a sender will not do this, as this puts himself at the same risk as every other intermediary. For a way of preventing this race entirely, we defer the reader to Appendix C.4.

Obfuscate the length of the path. By adding additional dummy outputs (that belong to fresh addresses of the sender) to $\mathsf{tx}^{\mathsf{er}}$, a sender can obfuscate the path length. Note that the rList has to include some random values as well so that it has the same number of elements as $\mathsf{tx}^{\mathsf{er}}$ has outputs. Note that by looking at the timelock in the LN, the path length or at least ones position within the path is leaked to some degree.

Extended privacy discussion. As mentioned in Section 4.4.1, Blitz achieves sender, receiver, and path privacy, which provides a measure of privacy in the case of a successful payment. To hide the path from users observing $\mathsf{tx}^{\mathsf{er}}$, we use stealth addresses for the outputs of $\mathsf{tx}^{\mathsf{er}}$. This allows to have path privacy as defined in Section 4.4.1, where malicious intermediaries cannot determine the participants of the payment other than their direct neighbors. We stress that as in the LN, the stronger notion of relationship anonymity [MMSK+17] does not hold. Two users can link a payment by comparing the transaction $\mathsf{tx}^{\mathsf{er}}$ in Blitz, or the hash value in the LN.

To make an on-chain linking of the sender impossible, we require the input of $\mathsf{tx}^{\mathsf{er}}$ to be fresh and unlinkable to the sender. In practice, this can be achieved as follows. The sender creates off-chain an intermediary transaction $\mathsf{tx}^{\mathsf{in}}$ that spends from an output under the sender's control $\mathsf{tx}^{\mathsf{sdr}}$ to a newly generated address of the sender, never used before. Then, $\mathsf{tx}^{\mathsf{er}}$ uses this output with the new address of $\mathsf{tx}^{\mathsf{in}}$ as input. Since $\mathsf{tx}^{\mathsf{in}}$ is off-chain, users observing $\mathsf{tx}^{\mathsf{er}}$ are unable to link the payment to an on-chain identity. Again, this is due to inputs referring to a transaction hash plus an id of the output.

In the pessimistic case, these properties do not hold anymore. If the transactions go on-chain, they can be linked together by observing a shared transaction $\mathsf{tx}^{\mathsf{er}}$ or time T. The same holds true in the LN, where transactions that spend from an HTLC with the same hash value, can be linked.

Redundancy for improving throughput and latency. Routing a payment through a path can fail or be delayed due to unknown channel balances, offline or malicious users, or other reasons. Following Boomerang [BNT20], a sender can construct several redundant payments across several paths, that differ in one or more users. For this, the sender creates a transaction $\mathsf{tx}^{\mathsf{er}}$ for each of these redundant payments and forwards them. Intermediary users have to open a payment construction (build $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$) for every $\mathsf{tx}^{\mathsf{er}}$ that they receive.

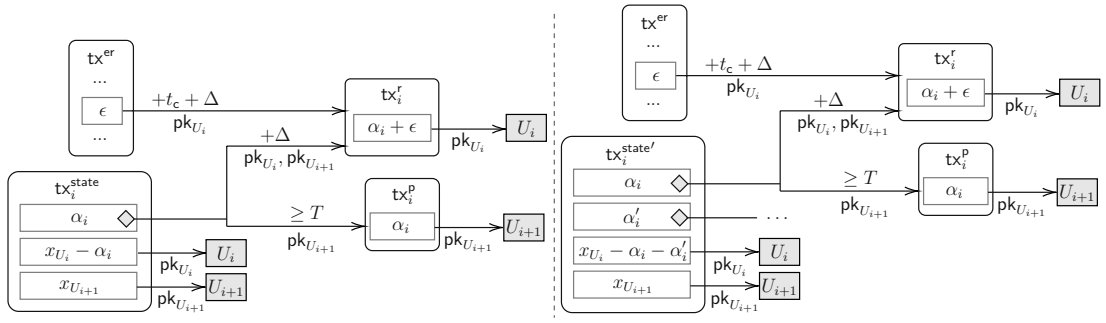

Figure C.1: Concurrent payments between users U_i and U_{i+1}: (left) a Blitz channel with a single payment; (right) an updated channel that has this payment and a second concurrent one. To add a second payment of value α_i' to the channel, the transactions for the in-flight payment of value α_i are recreated with the new state $\mathsf{tx}_i^{\mathsf{state}\prime}$ as input, the channel is updated to $\mathsf{tx}_i^{\mathsf{state}\prime}$ and finally, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked. In the LN, this process is the same, except that the HTLC contract and transactions are recreated, instead of the Blitz ones.

Should an intermediary user have a choice of forwarding a payment to several different neighbors, it can choose one and start a fast refund (Section 4.3) for the other payments. Should several different payments reach the receiver, it can start the fast refund for all but one of them. In the worst case, if the sender sees that after some time more than one payment is active, it can start the refund by publishing the according transaction $\mathsf{tx}^{\mathsf{er}}$. With this, the sender can ensure that at most one of the redundant payments is carried out. This technique is useful to improve transaction throughput and latency and we achieve it without any additional cryptography.

Concurrent payments. Two parties of a payment channel can achieve concurrent payments as follows. They agree to update their current channel state $\mathsf{tx}_i^{\mathsf{state}}$ to a new state $\mathsf{tx}_i^{\mathsf{state}\prime}$, where any unresolved in-flight Blitz payments are carried over. More concretely, for every unresolved payment the transactions $\mathsf{tx}_i^{\mathsf{r}}$ and $\mathsf{tx}_i^{\mathsf{p}}$ are recreated, but the input for these transactions is changed from using an output of $\mathsf{tx}_i^{\mathsf{state}}$ to using an output of $\mathsf{tx}_i^{\mathsf{state}\prime}$. Afterwards, the right user's signature for $\mathsf{tx}_i^{\mathsf{r}}$ is given to the left user and only then, the old state $\mathsf{tx}_i^{\mathsf{state}}$ is revoked using the revocation technique in the LN (outlined in Appendix C.3). In other words, the same channel state-management of the LN is reused in Blitz, but changing the HTLC contract for the Blitz contract. We show an illustrative example of concurrent payments in Figure C.1.

C.2 1-phase commits in distributed databases

The concepts of 1-phase commits [AhC95, SC93, AGP98] and one-two commit [AHC04] have been studied for distributed databases in general. These protocols introduce recovery mechanisms such as coordinator Log [SC93], implicit Yes-Vote [AhC95] or logical logging [AGP98] towards avoiding the voting/commit/prepare phase of 2-phase commits.

However, extending observation by Herlihy, Liskov, and Shrira [HSL19], traditional 1-phase commit ideas are not directly applicable to PCNs: while PCNs (with blockchain-based conflict resolution) are structurally similar to transactions over distributed database, they are fundamentally different in terms of the ACID properties and the adversarial assumptions. Nevertheless, analyses such as [GW17] can still be interesting to understand lower bounds for PCNs.

C.3 Payment channels in more detail

In this section, we give a more detailed account on payment channels. A payment channel is used by two parties P and Q to perform several payments between them while requiring only two on-chain transactions. It is set up by two parties spending some coins to a shared multisig output (i.e., an output θ with $\theta.\phi := \mathsf{MultiSig}(P, Q)$). Before signing and publishing this transaction, however, they create transactions (so-called *commitment transactions* tx^{c}) that spend this shared output in some way, e.g., giving each party some balance. We also refer to this as the (current) *state* of the channel. Now after publishing this tx^{f} on-chain, they can update their balances by creating new commitment transactions tx^{c}, rebalancing the funds of the channel, and thereby carrying out payments. We note that there are implementations that use two commitment transactions per state (in other words, one per party) such as the Lightning Network (LN) [PD16] whereas a more recent construction called generalized channels [AEE+21] requires one commitment transaction per state. In this work, we leverage the latter construction, although other ledger channel protocols such as the one of the LN would work as well.

After a channel has been updated several times, there exist several tx^{c} that can be published. In order to prevent misbehavior, where one party publishes an older state of the channel, which perhaps is financially more advantageous to it, we employ a punishment mechanism. If an old state is published, the other, honest user can carry out this punishment to gain all funds of the channel. For this to work, both parties exchange revocation secrets every time a state is succeeded by a new one. This secret, together with the outdated tx^{c} that is published by the misbehaving user is enough to claim all funds of the channel. The latest state can always be safely published as the corresponding revocation secret was not yet revealed. This mechanism provides an economical incentive not to publish an old tx^{c}.

To close a payment channel, the parties can merely publish the latest tx^{c} to the ledger, which terminates the channel. In summary, two parties can use a payment channel to carry out arbitrarily many off-chain payments that rebalance some funds, but only need to publish two transactions on the blockchain, one to open the channel and one to close it, saving both fees and increasing the cryptocurrency's transaction throughput.

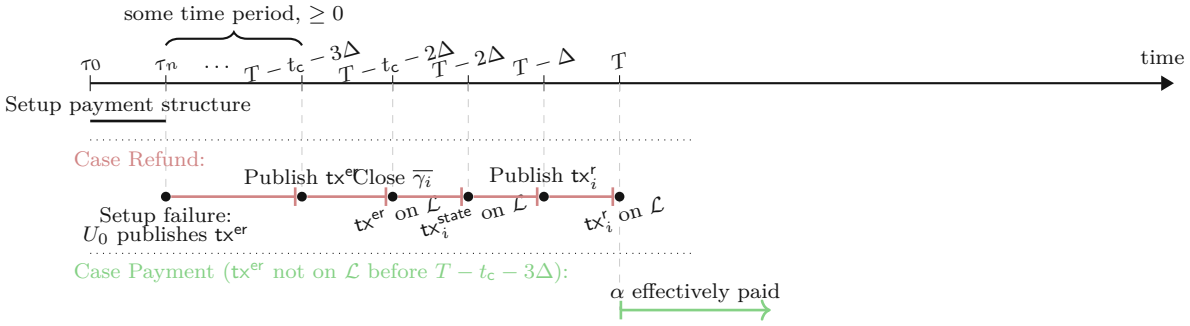

Figure C.2: Timeline of when transactions appear on the ledger $\mathcal{L}$ in the case payment and refund. $\tau_n - \tau_0$ denotes the time needed for the setup of the whole payment.

C.4 Preventing the race condition when the sender is irrational

We mentioned in Appendix C.1, that if the sender posts $\mathsf{tx}^{\mathsf{er}}$ after $T - t_{\mathsf{c}} - 3\Delta$ and before $T - t_{\mathsf{c}} - 2\Delta$, there is an unwanted race condition. We argue, that this race condition is also unwanted by the sender, but a small tweak to the protocol allows us to prevent it completely. We need to introduce a new spending condition to the output of $\mathsf{tx}^{\mathsf{in}}$, i.e., the output that is used to fund $\mathsf{tx}^{\mathsf{er}}$. Instead of the sender being able to just spend that output (and therefore $\mathsf{tx}^{\mathsf{er}}$) with the condition $\mathsf{OneSig}(\widetilde{U_0})$, we set the following condition: $(\mathsf{RelTime}(\Delta) \wedge \mathsf{OneSig}(\widetilde{U_0})) \vee \mathsf{AbsTime}(T - t_{\mathsf{c}} - 3\Delta)$. In other words, this output of $\mathsf{tx}^{\mathsf{in}}$ can be spent by anyone, if it is still unspent at time $T - t_{\mathsf{c}} - 3\Delta$.[2]

If U_0 wants to spend it, U_0 has to wait until the relative timelock of Δ has expired. So to be safely able to post $\mathsf{tx}^{\mathsf{er}}$, U_0 has to post $\mathsf{tx}^{\mathsf{in}}$ before $T - t_{\mathsf{c}} - 5\Delta$ (accepted to the ledger before $T - t_{\mathsf{c}} - 4\Delta$) and subsequently, $\mathsf{tx}^{\mathsf{er}}$ before $T - t_{\mathsf{c}} - 3\Delta$ (accepted to the ledger before $T - t_{\mathsf{c}} - 2\Delta$). After the relative timelock of $t_{\mathsf{c}} + \Delta$ expires, the outputs of $\mathsf{tx}^{\mathsf{er}}$ are spendable before $T - \Delta$, ensuring enough time and preventing a race. If $\mathsf{tx}^{\mathsf{in}}$ is posted later, then an observant intermediary can spend its output in a different way, which makes $\mathsf{tx}^{\mathsf{er}}$ unspendable. To convince intermediaries that this condition is actually present, the sender needs to pass (unsigned) $\mathsf{tx}^{\mathsf{in}}$ along with $\mathsf{tx}^{\mathsf{er}}$ along the path. To make $\mathsf{tx}^{\mathsf{in}}$ unlinkable to the sender, the same method mentioned in Appendix C.1 can be employed. We emphasize again, that this race is only a problem if we assume an irrational sender.

C.5 Concrete attack scenarios (informal)

In this section, we consider some attacks against Blitz and argue informally, why balance security still holds.

[2] In Bitcoin, an output can also be made spendable by anyone by putting OP_TRUE or requiring a signature that verifies under the private key 0x1, known by everybody.

$\mathsf{tx}^{\mathsf{er}}$ is tampered. If $\mathsf{tx}^{\mathsf{er}}$ is tampered by some intermediary, the next intermediary will see that the message embedded in the routing information is not $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ anymore. Assuming that a malicious intermediary does not know the routing information especially not the receiver, changing the routing information will result in the receiver not being reached.

Also, note that balance security holds even in the case where $\mathsf{tx}^{\mathsf{er}}$ is tampered, as long as every intermediary U_i makes sure, that its refund $\mathsf{tx}^{\mathsf{r}}_i$ depends on the same $\mathsf{tx}^{\mathsf{er}}$ as the refund of its neighbor $\mathsf{tx}^{\mathsf{r}}_{i-1}$. Also note, that intermediaries have to ensure the same for the time T, in order to have the same time as their neighbor. Should an intermediary change the time T to a smaller value, it potentially only hurts itself by not being able to refund in time, while its left neighbor actually is. If the time T is changed to a larger value, this may delay the execution of the payment, however, it is detectable, if the receiver sends this time T back to the sender, who can check if it was tampered.

Some users are skipped (wormhole). Users cannot be skipped, as the routing information can only be opened by the next user. A malicious user would not know the receiver and would not be able to forge the sender's signature of $\mathcal{H}(\mathsf{tx}^{\mathsf{er}})$ that is embedded as a message to the receiver in this onion. The only thing for the malicious user is to stop forwarding the payment (griefing attack). Users that are skipped in the fast-track payment will not be cheated out of their fees or funds, rather this money will be locked until at most until T instead of being accessible immediately (see Section 4.3).

Sender publishes $\mathsf{tx}^{\mathsf{er}}$ after starting fast track. Assume a malicious sender started the fast track with its neighbor, but the fast track updates have not yet reached the receiver. Should the sender now publish $\mathsf{tx}^{\mathsf{er}}$, the intermediaries that have not yet performed the fast track will refund. The receiver will say that it did not receive the money and will not ship the promised product. The sender cannot prove that the receiver got the money, even though it has the payment confirmation in form of the receiver's signature of $\mathsf{tx}^{\mathsf{er}}$. The transaction $\mathsf{tx}^{\mathsf{er}}$ on the blockchain is a proof of revocation, and the sender will have lost its money without getting anything in return. The sender should thus not publish $\mathsf{tx}^{\mathsf{er}}$ after starting the fast track.

C.6 Timeline

We show a timeline of posting the transaction of the Blitz payment construction between two users in Figure C.2. Red shows the refund case, green the payment case.

C.7 Communication overhead

To evaluate our payment scheme, we created an implementation that creates the transactions necessary for setting up the payment. The source code is publicly available at `https://github.com/blitz-payments/overhead`. We tested the compatibility by deploying the transactions on the Bitcoin testnet and checking if the transactions achieve our intended functionalities. Furthermore, we measured the transaction sizes in

Table C.1: Communication overhead of the LN and Blitz. The pessimistic transactions are on-chain, the rest off-chain.

Cases	LN		Blitz	
	# txs	size	# txs	size
Pay (pessimistic)	1	192	1	158
Refund (pessimistic) per channel	1	158	1	307
Additional pess. refund cost for sender	0	0	1	$157 + 34 \cdot n$
Cost of p in-flight payments	1	$225 + 119 \cdot p$	1	$225 + 88 \cdot p$

Bytes and compare them to multi-hop payments in the Lightning Network (LN) in a case-by-case analysis.

We present the number of transactions and their sizes for the different sizes in Table C.1. Note that the size of the contract in our construction is only 88 Bytes compared to the 119 of the HTLC, a difference mostly due to the part of the script that verifies the hash pre-image. This means, that state transactions holding several different in-flight payments, which directly implement the contract in their outputs, can hold around 26% more Blitz payments than LN payments. For one payment, this difference results in a state of size 311 Bytes for Blitz and a state of 345 Bytes for the LN. In Blitz, additionally to the state we require the refund transaction to be exchanged, which is 307 Bytes, resulting in 618 Bytes for a 2-party setup.

For the rest of the cases, the Blitz payments and the LN payments are similar. In the pessimistic case, both Blitz and the LN require to publish one transaction (after closing the channel) per disputed channel. In the pessimistic refund case, it is 158 Bytes in the LN and 307 Bytes in Blitz, due to the additional signature of the input spending from $\mathsf{tx}^{\mathsf{er}}$. In the pay case, it is 192 Bytes in the LN and 158 Bytes in Blitz, due to the additional hash in the LN. The most notable difference in comparing the transaction overhead comes from the fact that in the Blitz payment, the sender has to publish $\mathsf{tx}^{\mathsf{er}}$ in the pessimistic refund case, which is a total of $157 + 34 \cdot (n)$ Bytes, for a payment path of length $n + 1$. However, in the LN there is an additional communication overhead of sending the hash pre-image of 32 Bytes per channel back in the open phase.

C.8 Extended simulation results

In this section, we include results for the simulation when we do not distribute the disrupted payments equally between the two types. As expected, letting 75% of the disrupted payments be of the second type is more favorable for Blitz, while having 25% is less favoring than dividing equally. We show the results in Table C.2.

C.9 Extended macros

In this section, we give concrete pseudo-code for the used subprocedures.

Table C.2: Extended results of our simulation.

ub	*FRate*	ppnpr	$\mathsf{fail}_{\mathsf{Blitz}}$	$\mathsf{fail}_{\mathsf{LN}}$	ratio
25% disrupted type 1, 75% type 2					
3000	0.5%	4	4	33	8.25
3000	0.5%	50	13	4343	334.08
3000	1%	4	15	56	3.73
3000	1%	50	751	32807	43.68
3000	2.5%	4	28	182	6.50
3000	2.5%	50	1076	77213	71.76
75% disrupted type 1, 25% type 2					
3000	0.5%	4	18	31	1.72
3000	0.5%	50	505	4422	8.76
3000	1%	4	19	61	3.21
3000	1%	50	1458	33386	22.90
3000	2.5%	4	78	195	2.50
3000	2.5%	50	15427	77574	5.03

Subprocedures

`checkTxIn`($\mathsf{tx}^{\mathsf{in}}, n, U_0$):

1. Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n \cdot \epsilon$ and $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\phi = \mathsf{OneSig}(U_0')$, that is spendable by an unused address of U_0, return $\top$. Otherwise, return $\bot$. When using this transaction (to fund $\mathsf{tx}^{\mathsf{er}}$), the sender will pay any superfluous coins back to a fresh address of itself.

`checkChannels`($\mathsf{channelList}, U_0$):

Check that $\mathsf{channelList}$ forms a valid path from U_0 via some intermediaries to a receiver U_n and that no users are in the path twice. If not, return $\bot$. Else, return U_n.

`checkT`(n, T):

Let τ be the current round. If $T \geq \tau + n(2 + t_{\mathsf{u}}) + 3\Delta + t_{\mathsf{c}} + 1$, return $\top$. Otherwise, return $\bot$

`genTxEr`($U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}$):

1. Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$

2. For every channel γ_i in $\mathsf{channelList}$:
   - $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(\gamma_i.\mathsf{left}.A, \gamma_i.\mathsf{left}.B)$
   - $\mathsf{outputList} := \mathsf{outputList} \cup (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_\mathsf{c} + \Delta))$
   - $\mathsf{rList} := \mathsf{rList} \cup R_i$
3. Let $\mathcal{P} := \{\gamma_i.\mathsf{left},\ \gamma_i.\mathsf{right}\}_{\gamma_i \in \mathsf{channelList}}$ and let $\mathsf{nodeList}$ be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
4. Shuffle $\mathsf{outputList}$ and rList.
5. Let $\mathsf{tx}^{\mathsf{er}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$
6. Create a list $[\mathsf{msg}_i]_{i\in[0,n]}$, where $\mathsf{msg}_i := \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$
7. $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\mathsf{nodeList}, [\mathsf{msg}_i]_{i\in[0,n]})$
8. Return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion})$

`genState`$(\alpha_i, T, \overline{\gamma_i})$:

1. For the users $U_i := \overline{\gamma_i}.\mathsf{left} =$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$, create the output vector $\theta_i := (\theta_0, \theta_1, \theta_2)$, where
   - $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   - $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   - $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$

   where x_{U_i} and $x_{U_{i+1}}$ is the amount held by U_i and U_{i+1} in the channel, respectively.
2. Let $\mathsf{tx}_i^{\mathsf{state}}$ be a channel transaction carrying the state with $\mathsf{tx}^{\mathsf{state}}.\mathsf{output} = \theta_i$. Return $\mathsf{tx}_i^{\mathsf{state}}$.

`checkTxEr`$(U_i, a, b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i)$:

1. $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \perp$, return $\perp$. If U_i is the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{er}})$, return $(\top, \top, \top, \top, \top)$. Else, if $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{er}}), \mathsf{onion}_{i+1})$, return $\perp$.
2. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ it must hold that:
   - $\mathsf{cash} = \epsilon$
   - $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_\mathsf{c} + \Delta)$ for some identity pk_x
3. For exactly one output $\theta_{\epsilon_i} := (\epsilon,\ \mathsf{OneSig}(\widetilde{U_i}) \wedge \mathsf{RelTime}(t_\mathsf{c} + \Delta)) \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output}$ and one element $R_i \in \mathsf{rList}$ it must hold that
   - Let $\mathsf{pk}_{\widetilde{U_i}}$ be the corresponding public key of $\mathsf{OneSig}(\widetilde{U_i})$
   - $\mathsf{sk}_{\widetilde{U_i}} := \mathsf{GenSk}(a, b, \mathsf{pk}_{\widetilde{U_i}}, R_i)$ must be the corresponding secret key of $\mathsf{pk}_{\widetilde{U_i}}$
4. If the checks in 2 or 3 do not hold, return $\perp$
5. Return $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$

Subprocedures used exclusively in UC model

`createMaps`$(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}})$:

1. For every $U_i \in \mathsf{nodeList} \setminus U_n$ do:
   - $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(U_i.A, U_i.B)$
   - $\mathsf{outputMap}(U_i) := (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_c + \Delta))$
   - $\mathsf{rMap}(U_i) := R_i$
2. $\mathsf{rList} = \mathsf{rMap}$.`values()`.`shuffle()`
3. $\mathsf{tx}^{\mathsf{er}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputMap}$.`values()`.`shuffle()`$)$
4. Create a map stealthMap that stores for every user U_i that is a key in outputMap the corresponding output of $\mathsf{tx}^{\mathsf{er}}$ corresponding to $\mathsf{outputMap}(U_i)$
5. Create two empty lists $\emptyset$ named msgList, userList
6. For every $U_i \in \mathsf{nodeList}$ from U_n to U_0 (in descending order):
   - Append $[\mathcal{H}(\mathsf{tx}^{\mathsf{er}})]$ to msgList
   - Prepend $[U_i]$ to userList.
   - $\mathsf{onion}_i := \mathsf{CreateRoutingInfo}(\mathsf{userList}, \mathsf{msg})$
   - $\mathsf{onions}(U_i) := \mathsf{onion}_i$
7. Return $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap})$

`genStateOutputs`$(\overline{\gamma_i}, \alpha_i, T)$:

1. Let $\theta'_i := \overline{\gamma_i}.\mathsf{st}$ be the current state of the channel $\overline{\gamma_i}$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ = and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. θ'_i consists of the outputs $\theta'_{U_i} := (x_{U_i}, \ \mathsf{OneSig}(U_i))$ and $\theta'_{U_{i+1}} := (x_{U_{i+1}}, \ \mathsf{OneSig}(U_{i+1}))$ holding the balances of the two users.[a] If $x_{U_i} < \alpha_i$, return $\perp$
4. Create the output vector $\theta_i := (\theta_0, \theta_1, \theta_2)$, where
   - $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   - $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   - $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$
5. Return θ_i.

`genRefTx`$(\theta, \theta_{\epsilon_i}, U_i)$:

1. Create a transaction $\mathsf{tx}_i^{\mathsf{r}}$ with $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{input} := [\theta, \quad \theta_{\epsilon_i}]$ and $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\theta.\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))$.
2. Return $\mathsf{tx}_i^{\mathsf{r}}$

`genPayTx`(θ, U_{i+1}):

1. Create a transaction $\mathsf{tx}_i^{\mathsf{p}}$ with $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{input} := [\theta]$ and $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{output} := (\theta.\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$.
2. Return $\mathsf{tx}_i^{\mathsf{p}}$

[a] Possibly other outputs $\{\theta'_j\}_{j\geq 0}$ could also be present in this state. They, along with the off-chain objects there (e.g., other payments) would have to be recreated in the new state while adapting the index of the output these objects are referring to. For simplicity, we say this here in prose and omit it in the protocol, only handling the two outputs mentioned.

C.10 Modeling in the UC framework

We formally model our construction in the global UC framework (GUC) [CDPW07], an extension of the standard UC framework [Can01] that allows for a global setup, which we use for instance for modeling the ledger. In this section, we provide some preliminaries and then present the code for the ideal functionality of the multi-hop payment construction presented in this work. Our model follows closely the model in [AEE+21].

C.10.1 Preliminaries, communication model and threat model

A protocol Π runs between parties of the set $\mathcal{P}$. A protocol is executed in the presence of an adversary $\mathcal{A}$ that receives as input a security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0,1\}^*$. We assume a static corruption model, where $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the execution, which means learning P_i's internal state and taking full control over P_i. The environment $\mathcal{E}$ is a special entity, that sends inputs to every party and the adversary $\mathcal{A}$ and observes every message output by the parties. Note that $\mathcal{E}$ is used to model anything that can happen outside the protocol execution.

We model communication in a synchronized network setting, where the protocol execution takes place in rounds. This abstraction allows for arguing about time more naturally. The global ideal functionality $\mathcal{G}_{clock}$ [KMTZ13] represents a global clock, that proceeds to the next round when all honest parties agree to do so. Every entity is aware of what the current round is.

On top of this notion of rounds, we use a functionality $\mathcal{F}_{GDC}$ that models authenticated channels with guaranteed delivery after one round between the parties. Messages sent from a party P to Q in round t are guaranteed to reach Q in round $t+1$ with Q knowing that the sender was P. The adversary $\mathcal{A}$ can observe the content of messages and reorder the ones that were sent within the same round. It cannot however drop, modify, or delay messages. See [DEF+19b] for a formal description of $\mathcal{F}_{GDC}$.

Every other message, that is not sent between two parties, but rather involves for instance $\mathcal{E}$ or $\mathcal{A}$, takes zero rounds. Also, we assume that any computation by any party takes zero rounds as well.

C.10.2 Ledger and channels

To model a UTXO cryptocurrency, we use a global functionality $\mathcal{G}_{Ledger}(\Delta)$, parameterized by an upper bound of the blockchain delay Δ, i.e., the number of rounds it at most takes for a valid transaction to be accepted on the blockchain, after being posted, and a signature scheme Σ. This functionality interacts with a fixed set of parties $\mathcal{P}$. To initialize $\mathcal{G}_{Ledger}$, $\mathcal{E}$ sets up a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ for every $P \in \mathcal{P}$, sends $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_p)$ to $\mathcal{G}_{Ledger}$ and sets the intial state of $\mathcal{L}$, the set of all published transactions. After the initialization, the state of $\mathcal{L}$ is publicly accessible by every entity. When a valid transaction (i.e., a transaction that has correct witnesses for each input, a unique id, and the inputs have not been spent) is posted via $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}})$, it will be accepted on $\mathcal{L}$ after at most Δ rounds. The adversary chooses the exact number of rounds.

In this simplified model, the set of users is fixed and we do not model the fact, that in reality, transactions are usually bundled in blocks. We chose this simplification to increase readability and refer to works such as [BMTZ17] for a more accurate formalization.

To model channels, we use the functionality $\mathcal{F}_{Channel}$ [AEE+21] that builds on top of $\mathcal{G}_{Ledger}$. It provides the functionality to create, update, and close a payment channel between two users. We say that updating a channel takes at most t_{u} rounds and closing a channel, regardless if the parties are cooperating or not, takes at most t_{c} rounds.

For our Blitz payments, we assume that all participating parties have been registered with the ledger functionality and have had channels created beforehand already. For the complete API of $\mathcal{F}_{Channel}$ and $\mathcal{G}_{Ledger}$ see below. For better readability, we use the following notation instead of calling $\mathcal{G}_{clock}$ or $\mathcal{F}_{GDC}$. We let $(\mathsf{msg}) \overset{t}{\hookrightarrow} X$ denote sending message (msg) to X in round t. Moreover, $(\mathsf{msg}) \overset{t}{\hookleftarrow} X$ means receiving message (msg) from X at time t. Note that X as well as the sending/receiving identity are either a party $P \in \mathcal{P}$, the environment $\mathcal{E}$, the simulator $\mathcal{S}$ or another ideal functionality.

Interface of $\mathcal{F}_{Channel}(T, k)$ [AEE+21]

Parameters:

- T: upper bound on the maximum number of consecutive off-chain communication rounds between channel users
- k: number of ways the channel state can be published on the ledger

API:

Messages from $\mathcal{E}$ via a dummy user P:

- $(\texttt{sid}, \texttt{CREATE}, \overline{\gamma}, \mathsf{tid}_P) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the attribute tuple $(\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users}, \overline{\gamma}.\mathsf{cash}, \overline{\gamma}.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\overline{\gamma}.\mathsf{users} \subset \mathcal{P}$ are the users of the channel (and $P \in \overline{\gamma}.\mathsf{users}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is the total money in the channel and $\overline{\gamma}.\mathsf{st}$ is the initial state of the channel. tid_P defines P's input for the funding transaction of the channel. When invoked, this function asks $\overline{\gamma}.\mathsf{otherParty}$ to create a new channel.
- $(\texttt{sid}, \texttt{UPDATE}, \mathsf{id}, \theta) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the channel where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$. When invoked by $P \in \overline{\gamma}.\mathsf{users}$ and both parties agree, the channel $\overline{\gamma}$ (if it exists) is updated to the new state θ. If the parties disagree or at least one party is dishonest, the update can fail or the channel can be forcefully closed to either the old or the new state. Regardless of the outcome, we say that t_u is the upper bound that an update takes. In the successful case, $(\texttt{sid}, \texttt{UPDATED}, \mathsf{id}, \theta) \xhookrightarrow{\leq \tau + t_\mathsf{u}} \overline{\gamma}.\mathsf{users}$ is output.
- $(\texttt{sid}, \texttt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookleftarrow} P$:
  Will close the channel $\overline{\gamma}$, where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$, either peacefully or forcefully. After at most t_c in round $\leq \tau + t_\mathsf{c}$, a transaction tx with the current state $\overline{\gamma}.\mathsf{st}$ as output $(\mathsf{tx}.\mathsf{output} := \overline{\gamma}.\mathsf{st})$ appears on $\mathcal{L}$ (the public ledger of $\mathcal{G}_{Ledger}$).

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma)$ [AEE+21]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accpeted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions .

Parameters:

- Δ: upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$
- Σ: a digital signature scheme

API:

Messages from $\mathcal{E}$ via a dummy user $P \in \mathcal{P}$:

- $(\texttt{sid}, \texttt{REGISTER}, \mathsf{pk}_P) \overset{\tau}{\hookleftarrow} P$:
  This function adds an entry (pk_P, P) to PKI consisting of the public key pk_P and the user P, if it does not already exist.
- $(\texttt{sid}, \texttt{POST}, \overline{\mathsf{tx}}) \overset{\tau}{\hookleftarrow} P$:
  This function checks if $\overline{\mathsf{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

C.10.3 The UC-security definition

We denote Π as a *hybrid* protocol that accesses the ideal functionalities $\mathcal{F}_{\mathsf{prelim}}$ consisting of $\mathcal{F}_{Channel}$, $\mathcal{G}_{Ledger}$, $\mathcal{F}_{GDC}$ and $\mathcal{G}_{clock}$. An environment $\mathcal{E}$ that interacts with Π and an adversary $\mathcal{A}$ will on input a security parameter λ and an auxiliary input z output $\texttt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. Moreover, $\phi_{\mathcal{F}_{Pay}}$ denotes the ideal protocol of ideal functionality $\mathcal{F}_{Pay}$, where the dummy users simply forward their input to $\mathcal{F}_{Pay}$. It has access to the same

functionalities $\mathcal{F}_{\mathsf{prelim}}$. The output of $\phi_{\mathcal{F}_{Pay}}$ on input λ and z when interacting with $\mathcal{E}$ and a simulator $\mathcal{S}$ is denoted as $\text{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

If a protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, then any attack that is possible on the real-world protocol Π can be carried out against the ideal protocol $\phi_{\mathcal{F}_{Pay}}$ and vice versa. Our security definition is as follows.

Definition 9. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, w.r.t. $\mathcal{F}_{\mathsf{prelim}}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that we have

$$\left\{\text{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\text{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

where $\approx^c$ denotes computational indistinguishability.

C.10.4 Ideal functionality

In this section, we will describe the ideal functionality (IF) $\mathcal{F}_{Pay}$ in prose. We are only interested in protocols that realize this IF and never output an ERROR. For cases where ERROR is output, any guarantees are lost. These cases are not meaningful to us, they occur for instance when a transaction does not appear on the ledger as it should. We use the subprocedures defined in Appendix C.9. We divide the ideal functionality into three main parts: (i) Pay, (ii) Finalize and (iii) Respond.

Pay. This sequence starts with setup, which is executed when queried by the sender U_0. In it, $\mathcal{F}_{Pay}$ sets up all initial objects and does the following. For every neighbor distinguish two cases: (i) the neighbor is honest, then $\mathcal{F}_{Pay}$ takes care of computing the objects and updating the channel or (ii) the neighbor is dishonest, then $\mathcal{F}_{Pay}$ instructs the simulator to simulate the view of the attacker. Should an attack ask an honest node, simulated by the simulator, to continue opening a payment with a legitimate request, the simulator will first let $\mathcal{F}_{Pay}$ perform Check and then Register. This process is repeated until the receiver is reached. At this point, the Finalize part starts.

Finalize. If U_0 is honest, $\mathcal{F}_{Pay}$ will expect a confirmation in the correct round, which is given either by itself if U_n is honest or sent by U_n via the simulator. If the confirmation is not well-formed or no confirmation is received in the correct round, $\mathcal{F}_{Pay}$ instructs the simulator to publish $\mathsf{tx}^{\mathsf{er}}$. In case that U_n is honest, but not U_0, either $\mathcal{F}_{Pay}$ via the simulator or the simulator directly will simulate the view to the attacker by constructing and sending the confirmation to U_0.

Respond. In this phase $\mathcal{F}_{Pay}$ reacts to transactions $\mathsf{tx}^{\mathsf{er}}$, that it has registered for payments in step Pay, appearing on $\mathcal{L}$. In the case of $\mathsf{tx}^{\mathsf{er}}$ for an honest user in a channel being published before the time when a refund is possible, $\mathcal{F}_{Pay}$ will close the channel and ask the simulator to publish a refund transaction. In the case that the time T has already passed and the neighbor closes the channel, $\mathcal{F}_{Pay}$ will instruct the simulator to claim the money by publishing the payment transaction.

Ideal Functionality $\mathcal{F}_{Pay}(\Delta)$

Parameters:

Δ : Upper bound on the time it takes a transaction to appear on $\mathcal{L}$.

Local variables:

idSet : A set of containing pairs of ids and users $(\texttt{pid}, U_i)$ to prevent duplicate ids to avoid loops in payments.

Φ : A map, storing for a given key $(\texttt{pid}, U_0)$ of an id pid and a user U_0, a tuple $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{er}}, U_n)$, where τ_{f} is the round in which the payment confirmation is expected from the receiver, the transaction $\mathsf{tx}^{\mathsf{er}}$ and the receiver U_n. The map is initially empty and read write access is written as $\Phi(\texttt{pid}, U_0)$. $\Phi.\texttt{keyList}()$ returns a set of all keys.

Γ : A set of tuples $(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R_i)$ for channels with opened payment construction, containing a payment id pid, the channel $\overline{\gamma_i}$, the state the payment builds upon θ_i, the time T, the output used in the refund by $\overline{\gamma_i}.\mathsf{left}$ and value R_i to reconstruct the secrect key of the stealth address used. It is initially empty.

Ψ : A set of tuples $(\texttt{pid}, \mathsf{tx}^{\mathsf{er}})$ containing payments, that have been opened and where the receiver is honest.

t_{u} : Time required to perform a ledger channel update honestly.

t_{c} : Time it at most takes to close a channel.

Init (executed at initialization in round t_{init}.)

Send $(\texttt{sid}, \texttt{init}) \xhookrightarrow{t_{\mathsf{init}}} \mathcal{S}$ and upon $(\texttt{sid}, \texttt{init-ok}, t_{\mathsf{u}}, t_{\mathsf{c}}) \xhookleftarrow{t_{\mathsf{init}}} \mathcal{S}$ set t_{u} and t_{c} accordingly.

Pay

Let τ be the current round.

Setup:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \xhookleftarrow{\tau} U_0$, if $(\texttt{pid}, U_0) \in \mathsf{idSet}$ go idle. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid}, U_0)\}$
2. Let $x := \texttt{checkChannels}(\mathsf{channelList}, U_0)$. If $x = \bot$, go idle. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in $\mathsf{channelList}$), go idle. Let $\mathsf{nodeList}$ be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, go idle.
4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, go idle.
5. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}})$.
6. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$.

7. Set $\Phi(\texttt{pid}, U_0) := (\tau_\mathsf{f} := \tau + n \cdot (2 + t_\mathsf{u}) + 1, \mathsf{tx}^{\mathsf{er}}, U_n)$.
8. If U_1 honest, execute **Open**$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_0, T, \overline{\gamma_0})$.
9. Else, let $\mathsf{onion}_1 := \mathsf{onions}(U_1)$ and $\theta_{\epsilon_0} := \mathsf{stealthMap}(U_0)$. Send $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, \alpha_0, T, \overline{\gamma_0}, \theta_{\epsilon_0}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

Continue:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. **Open**$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}})$.

Check:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{check-id}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. If $(\texttt{pid}, U_i) \notin \mathsf{idSet}$, let $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid}, U)\}$ and send the message $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \overset{\tau}{\hookrightarrow} U_i$
3. If $(\texttt{sid}, \texttt{pid}, \texttt{ACCEPT}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} U_i$, $(\texttt{sid}, \texttt{pid}, \texttt{ok}, \overline{\gamma_i}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

Payment-Open:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{payment-open}, \mathsf{tx}^{\mathsf{er}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$, let $\Psi := \Psi \cup \{(\texttt{pid}, \mathsf{tx}^{\mathsf{er}})\}$.

Register:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. $\Gamma := \Gamma \cup \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R)\}$

Open$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}})$:
Let τ be the current round and $U_i := \overline{\gamma_{i-1}}.\mathsf{right}$

1. If $(\texttt{pid}, U_i) \in \mathsf{idSet}$, go idle.
2. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\texttt{pid}, U_i)\}$
3. If an entry after U_i in $\mathsf{nodeList}$ exists and is $\perp$, go idle.
4. If $U_i = U_n$ (i.e., last entry in $\mathsf{nodeList}$), set $U_{i+1} := \top$. Else, get U_{i+1} from $\mathsf{nodeList}$ (the entry after U_i).
5. $R_i := \mathsf{rMap}(U_i)$ and $\theta_{\epsilon_i} := \mathsf{stealthMap}(U_i)$
6. $\theta_{i-1} := \texttt{genStateOutputs}(\overline{\gamma_{i-1}}, \alpha_{i-1}, T)$. If $\theta_{i-1} = \perp$, go idle. Else, wait 1 round.
7. $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \overset{\tau+1}{\longrightarrow} U_i$
8. If not $(\texttt{sid}, \texttt{pid}, \texttt{ACCEPT}, \overline{\gamma_i}) \overset{\tau+1}{\longleftarrow} U_i$, go idle. Else,wait 1 round.
9. $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_{i-1}}.\mathsf{id}, \theta_{i-1}) \overset{\tau+2}{\longrightarrow} \mathcal{F}_{Channel}$

10. $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}) \xleftarrow{\tau+2+t_u} \mathcal{F}_{Channel}$, else go idle.
11. $\Gamma := \Gamma \cup (\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R_i)$
12. If $U_i = U_n$:
    - $\Psi := \Psi \cup \{(\texttt{pid}, \mathsf{tx}^{\mathsf{er}})\}$
    - $(\texttt{sid}, \texttt{pid}, \texttt{PAYMENT-OPEN}, \mathsf{tx}^{\mathsf{er}}, T, \alpha_{i-1}) \xrightarrow{\tau+2+t_u} U_i$
    - If U_0 is dishonest, send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}) \xrightarrow{\tau+2+t_u} \mathcal{S}$
13. Else:
    - $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xrightarrow{\tau+2+t_u} U_i$
    - If U_{i+1} honest, execute **Open**$(\texttt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1} - \mathsf{fee}, \overline{\gamma_i})$
    - Else, send $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_i}, \theta_{\epsilon_i}) \xhookrightarrow{\tau} \mathcal{S}$, where $\mathsf{onion}_{i+1} := \mathsf{onions}(U_{i+1})$

Finalize (executed at every round)

For every $(\texttt{pid}, U_0) \in \Phi.\texttt{keyList}()$ do the following:

1. Let $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{er}}, U_n) = \Phi(\texttt{pid}, U_0)$. If for the current round τ it holds that $\tau = \tau_f$, do the following.
2. If U_n honest, check if $(\texttt{pid}, \mathsf{tx}^{\mathsf{er}}) \in \Psi$. If yes, let $\Psi := \Psi \setminus \{(\texttt{pid}, \mathsf{tx}^{\mathsf{er}})\}$ and go idle.
3. If U_n dishonest and $(\texttt{sid}, \texttt{pid}, \texttt{confirmed}, \mathsf{tx}_x^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}_x^{\mathsf{er}})) \xhookleftarrow{\tau_{\mathsf{f}}} \mathcal{S}$, such that $\mathsf{tx}_x^{\mathsf{er}} = \mathsf{tx}^{\mathsf{er}}$ and $\sigma_{U_n}(\mathsf{tx}_x^{\mathsf{er}})$ is U_n's valid signature of $\mathsf{tx}^{\mathsf{er}}$, go idle.
4. Send $(\texttt{sid}, \texttt{pid}, \texttt{denied}, \mathsf{tx}^{\mathsf{er}}, U_0) \xhookrightarrow{\tau_{\mathsf{f}}} \mathcal{S}$. $\mathsf{tx}^{\mathsf{er}}$ must appear on $\mathcal{L}$ in round $\tau' \leq \tau_{\mathsf{f}} + \Delta$. Otherwise, output $(\texttt{sid}, \texttt{ERROR}) \xhookrightarrow{t_1} U_0$.

Respond (executed at the end of every round)

Let t be the current round. For every element $(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R_i) \in \Gamma$, check if $\overline{\gamma_i}.\mathsf{st} = \theta_i$ and $\mathsf{tx}^{\mathsf{er}}$ is on $\mathcal{L}$. If yes, do the following:

Revoke: If $\gamma_i.\mathsf{left}$ honest and $t < T - t_{\mathsf{c}} - 2\Delta$ do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R_i)\}$.
- $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \xhookrightarrow{t} \mathcal{F}_{Channel}$
- At time $t + t_{\mathsf{c}}$, a transaction tx with $\mathsf{tx}.\mathsf{output} = \overline{\gamma_i}.\mathsf{st}$ has to be on $\mathcal{L}$. If not, do the following. If $(\texttt{ssid}_C, \texttt{PUNISHED}, \overline{\gamma_i}.\mathsf{id}) \xleftarrow{\tau<T} \mathcal{F}_{Channel}$, go idle. Else, send $(\texttt{sid}, \texttt{ERROR}) \xhookrightarrow{T} \gamma_i.\mathsf{users}$.

- Wait for Δ rounds and send $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma_i}, \theta_{\epsilon_i}, R_i) \xhookrightarrow{t'<T-\Delta} \mathcal{S}$
- At time $t'' < T$, check whether a transaction tx' appears on $\mathcal{L}$ with $\mathsf{tx}'.\mathsf{input} = [\theta_{\epsilon_i}, tx.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = [(\mathsf{tx.output}[0].\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))]$. If it does, send $(\texttt{sid}, \texttt{pid}, \texttt{REVOKED}) \xhookrightarrow{t''} \overline{\gamma_i}.\mathsf{left}$. If not, send $(\texttt{sid}, \texttt{ERROR}) \xhookrightarrow{T} \gamma_i.\mathsf{users}$.

Force-Pay: Else, if a transaction tx with $\mathsf{tx.output} = \overline{\gamma_i}.\mathsf{st}$ is on-chain and $\mathsf{tx.output}[0]$ is unspent (i.e., there is no transaction on $\mathcal{L}$, that uses is as input), $t \geq T$ and U_{i+1} is honest, do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R_i)\}$.
- Send $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma_i}) \xhookrightarrow{t} \mathcal{S}$
- In round $t + \Delta$ transaction tx' with $\mathsf{tx}'.\mathsf{input} = [\mathsf{tx.output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = (\mathsf{tx.output}[0].\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$ must have appeared on $\mathcal{L}$. If yes, $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY}) \xhookrightarrow{t+\Delta} \overline{\gamma_i}.\mathsf{right}$. Otherwise, $(\texttt{sid}, \texttt{ERROR}) \xhookrightarrow{t+\Delta} \gamma_i.\mathsf{users}$.

C.10.5 Protocol

Here we present the formal protocol Π and a brief description thereof. For simplicity, we assume that users involved in the payment do not use (e.g., update, close) the channels involved in the payment.[3] Moreover, for any payment the sender knows the receiver and the receiver knows the sender. Also, every user knows if it is the sender in a payment or if it is the receiver in a payment. Therefore, when the simulator simulates the behavior of an honest user, the simulator also knows if the user is the sender/receiver or not and, if it is the sender (receiver), the simulator also knows the receiver (sender).

The protocol itself is similar to the simpler version presented in Section 4.4.4, but extended with payment ids and UC formalism, most notably we introduce rounds and the environment $\mathcal{E}$. To reiterate briefly, the protocol is divided into three parts. In Pay, the initial objects are set up by U_0 after being invoked by $\mathcal{E}$. Afterwards, the neighbor is contacted and they open a payment construction by creating a new state, the appropriate transactions, signing them, and then updating the channel. When first asked, a user will forward an open message to $\mathcal{E}$, which responds with accept (or nothing). In Finalize, the receiver sends a confirmation to the sender. The sender expects the correct confirmation in the correct round, otherwise, it will publish $\mathsf{tx}^{\mathsf{er}}$. In the Respond phase, users will react to $\mathsf{tx}^{\mathsf{er}}$ being published and, if possible, either refund or force the payment.

Protocol Π

Let $\mathsf{fee} \in \mathbb{N}$ be a system parameter known to every user.

Local variables of U_i (all initially empty):

[3] We refer the reader to Appendix C.1 for an outline on how to perform concurrent payments or use the channel otherwise while a payment is active.

pidSet :	A set storing every payment id pid that a user has participated in to prevent duplicates.
paySet :	A map storing tuples $(\texttt{pid}, \tau_{\mathsf{f}}, U_n)$ where pid is an id, τ_{f} is the round in which a confirmation is expected from the receiver U_n for the payments that have been opened by this user.
local :	A map, storing for a given pid U_i's local copy of $\mathsf{tx}^{\mathsf{er}}$ and T in a tuple $(\mathsf{tx}^{\mathsf{er}}, T)$.
left :	A map, storing for a given pid a tuple $(\overline{\gamma_{i-1}}, \theta_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1})$ containing channel with its left neighbor U_{i-1}, the state and the transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ for U_i's left channel in the payment pid.
right :	A map, sotring for a given pid a tuple $(\overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U_i}})$ containing the channel with its right neighbor, the state, the transaction $\mathsf{tx}^{\mathsf{r}}_i$ and the key necessary for signing the refund transaction in the payment pid.
rightSig :	A map, storing for a given pid the signature for $\mathsf{tx}^{\mathsf{r}}_i$ of the right neighbor $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ in the payment pid.

Pay

Setup: In every round, every node $U_0 \in \mathcal{P}$ does the following. We denote τ_0 as the current round.

U_0 upon $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$

1. If pid $\in$ pidSet, abort. Add pid to pidSet.
2. Let $x :=$ checkChannels(channelList, U_0). If $x = \bot$, abort. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in channelList), go idle. Let nodeList be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, abort.
4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$, abort
5. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}})$.
6. $(\mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_0) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
7. $\mathsf{paySet} := \mathsf{paySet} \cup \{(\texttt{pid}, \tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 1, U_n)\}$
8. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) := \texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_0)$
9. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{er}}, T)$.
10. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n - 1)$ and compute:
    - $\theta_0 := \texttt{genStateOutputs}(\overline{\gamma_0}, \alpha_0, T)$
    - $\mathsf{tx}^{\mathsf{r}}_0 := \texttt{genRefTx}(\theta_0[0], \theta_{\epsilon_0}, U_0)$
11. Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_0}, \theta_0, \mathsf{tx}^{\mathsf{r}}_0, \mathsf{sk}_{\widetilde{U_0}})$.
12. Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_1, \theta_0, \mathsf{tx}^{\mathsf{r}}_0) \overset{\tau_0}{\hookrightarrow} U_1$.

Open: In every round, every node $U_{i+1} \in \mathcal{P}$ does the following. We denote τ_x as the current round.

U_{i+1} u. $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}_i^{\mathsf{r}}) \xleftarrow{\tau_x} U_i$

1. Perform the following checks:
   - Verify that $\texttt{pid} \notin \mathsf{pidSet}$. Add $\texttt{pid}$ to pidSet
   - Let $x := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$. Check that $x \neq \perp$, but instead $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$.
   - Set $\alpha_i = \theta_i[0].\mathsf{cash}$ and extract T from $\theta_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$).
   - Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\theta_i = \texttt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$.
   - Check that $\mathsf{tx}_i^{\mathsf{r}} := \texttt{genRefTx}(\theta_i[0], \theta_{\epsilon_x}, U_i)$, where θ_{ϵ_x} is an output of $\mathsf{tx}^{\mathsf{er}}$ that is not $\theta_{\epsilon_{i+1}}$.
2. If one or more of the previous checks fail, abort. Otherwise, send $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T) \xhookrightarrow{\tau_x} \mathcal{E}$.
3. If $(\texttt{sid}, \texttt{pid}, \texttt{ACCEPT}, \overline{\gamma_{i+1}}) \xhookleftarrow{\tau_x} \mathcal{E}$, generate $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$. Otherwise stop.
4. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}_i^{\mathsf{er}}, T)$, $\mathsf{left}(\texttt{pid}) := (\overline{\gamma_i}, \theta_i, \mathsf{tx}_i^{\mathsf{r}})$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})) \xhookrightarrow{\tau_x} U_i$.

U_i upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})) \xhookleftarrow{\tau_i+2} U_{i+1}$

(The round τ_i given U_i and $\texttt{pid}$ is defined in Setup or in Open step (6), the round when the update is successful.)

5. Check that $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is a valid signature for $\mathsf{tx}_i^{\mathsf{r}}$. If yes, set $\mathsf{rightSig}(\texttt{pid}) := \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ and $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xhookrightarrow{\tau_i+2} \mathcal{F}_{Channel}$.

U_{i+1} upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xhookleftarrow{\tau_x+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$

6. Define $\tau_{(i+1)} := \tau_x + 1 + t_{\mathsf{u}}$.
7. If U_{i+1} is not the receiver, using the values of step 1:
   - Send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xhookrightarrow{\tau_{i+1}} \mathcal{E}$.
   - $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}_i^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$
   - $\theta_{i+1} := \texttt{genStateOutputs}(\overline{\gamma_{i+1}}, \alpha_i - \mathsf{fee}, T)$

- $\mathsf{tx}^{\mathsf{r}}_{i+1} := \mathtt{genRefTx}(\theta_{i+1}[0], \theta_{\epsilon_{i+1}}, U_{i+1})$
- Set $\mathsf{right}(\mathtt{pid}) := (\widetilde{\gamma_{i+1}}, \theta_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}, \mathsf{sk}_{\widetilde{U_{i+1}}})$
- Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+2}, \theta_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}) \overset{\tau_{i+1}}{\hookrightarrow} U_{i+2}$.

8. If U_{i+1} is the receiver:
   - $\mathsf{msg} := \mathsf{GetRoutingInfo}(\mathsf{onion}_{i+1}, U_{i+1})$
   - Create the signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_i)$ as confirmation and send $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau_{i+1}}{\hookrightarrow} U_0$. Send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx}^{\mathsf{er}}, T, \alpha_i) \overset{\tau_{i+1}}{\hookrightarrow} \mathcal{E}$.

Finalize

U_0 in every round τ

For every entry $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n) \in \mathsf{paySet}$ do the following if $\tau = \tau_{\mathsf{f}}$:

1. Remove $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n)$ from paySet.
2. Upon receiving $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookleftarrow} U_n$, continue if $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ is a valid signature for $\mathsf{tx}^{\mathsf{er}}$. Otherwise, go to step (4).
3. If $\mathsf{local}(\mathtt{pid}) = \mathsf{tx}^{\mathsf{er}}$, go idle. Otherwise, continue with the next step.
4. Sign $\mathsf{tx}^{\mathsf{er}}$ yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})$ and set $\overline{\mathsf{tx}^{\mathsf{er}}} := (\mathsf{tx}^{\mathsf{er}}, (\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})))$. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{er}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Respond

U_i at the end of every round

Let t be the current round. Do the following:

1. For every $\mathtt{pid}$ in $\mathsf{right}.\mathtt{keyList}()$, let $(\overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U_i}}) := \mathsf{right}(\mathtt{pid})$, let $(\mathsf{tx}^{\mathsf{er}}, T) := \mathsf{local}(\mathtt{pid})$ and do the following. If $t < T - t_{\mathsf{c}} - 2\Delta$, $\mathsf{tx}^{\mathsf{er}}$ is on the ledger $\mathcal{L}$ and $\overline{\gamma_i}.\mathsf{st} = \theta_i$, do the following:
   - Remove the entry for $\mathtt{pid}$ from right, send $(\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$.
   - If a transaction tx with $\mathsf{tx}.\mathsf{output} = \theta_i$ is on $\mathcal{L}$ in round $t + t_{\mathsf{c}}$, wait Δ rounds.
   - Sign $\mathsf{tx}^{\mathsf{r}}_i$ yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i)$ and use $\mathsf{sk}_{\widetilde{U_i}}$ to sign $\mathsf{tx}^{\mathsf{r}}_i$ yielding $\sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)$.
   - Set $\overline{\mathsf{tx}^{\mathsf{r}}_i} := (\mathsf{tx}^{\mathsf{r}}_i, (\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i), \mathsf{rightSig}(\mathtt{pid}), \sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)))$ and send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i}) \overset{t+t_{\mathsf{c}}+\Delta}{\hookrightarrow} \mathcal{G}_{Ledger}$. When it appears on $\mathcal{L}$ in round $t_1 < T$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED}) \overset{t_2}{\hookrightarrow} \mathcal{E}$

2. For every pid in $\mathsf{left}.\texttt{keyList}()$, let $(\overline{\gamma_{i-1}},\ \theta_{i-1},\ \mathsf{tx}^{\mathsf{r}}_{i-1}) := \mathsf{left}(\texttt{pid})$, let $(\mathsf{tx}^{\mathsf{er}},\ T) := \mathsf{local}(\texttt{pid})$ and do the following. If $t \geq T$ and a transaction tx with $\mathsf{tx}.\mathsf{output} = \theta_{i-1}$ is on the ledger $\mathcal{L}$, but not $\mathsf{tx}^{\mathsf{r}}_{i-1}$, do the following:
   - Remove the entry for pid from left and create $\mathsf{tx}^{\mathsf{p}}_{i-1} := \texttt{genPayTx}(\overline{\gamma_{i-1}}.\mathsf{st}, U_i)$.
   - Sign $\mathsf{tx}^{\mathsf{p}}_{i-1}$ yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1})$.
   - Set $\overline{\mathsf{tx}^{\mathsf{p}}_{i-1}} := (\mathsf{tx}^{\mathsf{p}}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1}))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i-1}}) \overset{t}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   - If it appears on $\mathcal{L}$ in round $t_1 \leq t + \Delta$, send $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY}) \overset{t_1}{\hookrightarrow} \mathcal{E}$

C.10.6 Proof

In this section, we describe the simulator as well as the formal proof that the Blitz protocol (see Appendix C.10.5) UC-realizes the ideal functionality $\mathcal{F}_{Pay}$ shown in Appendix C.10.4.

Simulator

Local variables:

right	A map, storing the transaction $\mathsf{tx}^{\mathsf{r}}_i$ for a given keypair consisting of a payment id pid and a user U_i.
rightSig	A map, storing the signature of the right neighbor for the transaction stored in right for a given keypair consisting of a payment id pid and a user U_i.

Simulator for init phase

Upon $(\texttt{sid}, \texttt{init}) \overset{t_{\mathsf{init}}}{\hookleftarrow} \mathcal{F}_{Pay}$ and send $(\texttt{sid}, \texttt{init-ok}, t_{\mathsf{u}}, t_{\mathsf{c}}) \overset{t_{\mathsf{init}}}{\hookrightarrow} \mathcal{F}_{Pay}$.

Simulator for pay phase

a) Case U_i is honest, U_{i+1} dishonest

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\epsilon_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\epsilon_i})$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. $\theta_i := \texttt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$
4. $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\theta_i[0], \theta_{\epsilon_i}, U_i)$
5. $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i) \overset{\tau}{\hookrightarrow} U_{i+1}$
6. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \overset{\tau+2}{\hookleftarrow} U_{i+1}$, check that $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ is a valid signature for $\mathsf{tx}^{\mathsf{r}}_i$. If yes, set $\mathsf{rightSig}(\texttt{pid}, U_i) := \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$, $\mathsf{right}(\texttt{pid}, U_i) := \mathsf{tx}^{\mathsf{r}}_i$ and $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \overset{\tau+2}{\hookrightarrow} \mathcal{F}_{Channel}$. Send$(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{er}}, T, \theta_{\epsilon_i}, R) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$. Otherwise, go idle.

b) Case U_i is honest, U_{i-1} dishonest

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i, \theta_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1}) \xleftarrow{\tau} U_{i-1}$. Let $\alpha_{i-1} := \theta_{i-1}[0].\mathsf{cash}$ and extract T from $\theta_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$).
2. Let $x := \texttt{checkTxEr}(U_i, U_i.a, U_i.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_i)$. Check that $x \neq \bot$, but instead $x = (\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$. Otherwise, go idle.
3. Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\theta_{i-1} = \texttt{genStateOutputs}(\overline{\gamma_{i-1}}, \alpha_{i-1}, T)$ and that $\mathsf{tx}^{\mathsf{r}}_{i-1} := \texttt{genRefTx}(\theta_{i-1}[0], \theta_{\epsilon_x}, U_{i-1})$ for an output $\theta_{\epsilon_x} \in \mathsf{tx}^{\mathsf{er}}.\mathsf{output} \neq \theta_{\epsilon_i}$.
4. $(\texttt{sid}, \texttt{pid}, \texttt{check-id}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$
5. If not $(\texttt{sid}, \texttt{pid}, \texttt{ok}, \overline{\gamma_i}) \xleftarrow{\tau} \mathcal{F}_{Pay}$, go idle. Let $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
6. Sign $\mathsf{tx}^{\mathsf{r}}_{i-1}$ on behalf of U_i yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
7. Upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}, \theta_{i-1}) \xleftarrow{\tau+1+t_{\mathrm{u}}} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_{i-1}}, \theta_{i-1}, \mathsf{tx}^{\mathsf{er}}, T, \bot, \bot) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$. Otherwise, go idle.
8. If $U_i = U_n$ (if $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1}) = (\top, \top, \top, \top, \top)$ holds), and U_0 is honest,[a] send $(\texttt{sid}, \texttt{pid}, \texttt{payment-open}, \mathsf{tx}^{\mathsf{er}}) \xhookrightarrow{\tau+1+t_{\mathrm{u}}} \mathcal{F}_{Pay}$. If U_0 is dishonest, create signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$ on behalf of U_n and send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \xhookrightarrow{\tau+1+t_{\mathrm{u}}} U_0$. In both cases, send via $\mathcal{F}_{Pay}$ to the dummy user U_n the message $(\texttt{sid}, \texttt{pid}, \texttt{PAYMENT-OPEN}, \mathsf{tx}^{\mathsf{er}}, T, \alpha_{i-1}) \xhookrightarrow{\tau+1+t_{\mathrm{u}}} U_n$.
9. Send via $\mathcal{F}_{Pay}$ to the dummy user U_i the message $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xhookrightarrow{\tau+1+t_{\mathrm{u}}} U_i$.
10. If U_{i+1} honest, call process$(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_i, T)$
11. If dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_i}, \theta_{\epsilon_i})$.

process$(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{er}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_{i-1}, T)$

Let τ be the current round.

1. Initialize $\mathsf{nodeList} := \{U_i\}$ and $\mathsf{onions}, \mathsf{rMap}, \mathsf{stealthMap}$ as empty maps.
2. $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1}) := \mathsf{GetRoutingInfo}(\mathsf{onion}_i)$
3. $\mathsf{stealthMap}(U_i) := \theta_{\epsilon_i}$
4. $\mathsf{rMap}(U_i) := R_i$
5. While U_i and U_{i+1} honest:
   - $x := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1})$:
     - If $x = \bot$, append U_{i+1} and then $\bot$ to $\mathsf{nodeList}$ and break the loop.

- If $x = (\top, \top, \top, \top, \top)$, append U_{i+1} to $\mathsf{nodeList}$ and break the loop.
- Else, if $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, U_{i+2}, \mathsf{onion}_{i+2})$, do the following.

- Append U_{i+1} to $\mathsf{nodeList}$
- $\mathsf{onions}(U_{i+2}) := \mathsf{onion}_{i+2}$
- $\mathsf{rMap}(U_{i+1}) := R_{i+1}$
- $\mathsf{stealthMap}(U_{i+1}) := \theta_{\epsilon_{i+1}}$
- If U_{i+2} is dishonest, append U_{i+2} to $\mathsf{nodeList}$ and break the loop.
- Set $i := i + 1$ (i.e., continue loop for U_{i+1} and U_{i+2})

6. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{er}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$

[a] For simplicity, assume that the U_n (and in the case it is honest, the simulator) knows the sender. As the payment is usually tied to the exchange of some goods, this is a reasonable assumption. Note that in practice, this is not necessary, as the sender can be embedded in the routing information onion_n.

Simulator for finalize phase

a) Publishing $\mathsf{tx}^{\mathsf{er}}$

Upon receiving a message $(\mathtt{sid}, \mathtt{pid}, \mathtt{denied}, \mathsf{tx}^{\mathsf{er}}, U_0) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ and U_0 honest, sign $\mathsf{tx}^{\mathsf{er}}$ on behalf of U_0 yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{er}})$. Set $\overline{\mathsf{tx}^{\mathsf{er}}} := (\mathsf{tx}^{\mathsf{er}}, \sigma_{U_0}(\mathsf{tx}^{\mathsf{er}}))$ and send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{er}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

b) Case U_n honest, U_0 dishonest

Upon message $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{er}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$, sign $\mathsf{tx}^{\mathsf{er}}$ on behalf of U_n yielding $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})$. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookrightarrow} U_0$.

c) Case U_n dishonest, U_0 honest

Upon message $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookleftarrow} U_n$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmed}, \mathsf{tx}^{\mathsf{er}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}})) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$.

Simulator for respond phase

In every round τ, upon receiving the following two messages, react accordingly.

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{post\text{-}refund}, \overline{\gamma_i}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$.
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$.

- If U_{i+1} is honest, create the transaction $\mathsf{tx}_i^{\mathsf{r}} := \texttt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0],\ \theta_{\epsilon_i},\ U_i)$. Else, let $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{right}(\texttt{pid}, U_i)$
- Extract $\mathsf{pk}_{\widetilde{U}_i}$ from the output θ_{ϵ_i} of $\mathsf{tx}^{\mathsf{er}}$ and let $\mathsf{sk}_{\widetilde{U}_i} := \mathsf{GenSk}(U_i.a, U_i.b, \mathsf{pk}_{\widetilde{U}_i}, R_i)$.
- Generate signatures $\sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}})$ and, using $\mathsf{sk}_{\widetilde{U}_i}$, $\sigma_{\widetilde{U}_i}(\mathsf{tx}_i^{\mathsf{r}})$ on behalf of U_i.
- If $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$ is honest, generate signature $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ on behalf of U_{i+1}. Else, let $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}) := \mathsf{rightSig}(\texttt{pid}, U_i)$
- Set $\overline{\mathsf{tx}_i^{\mathsf{r}}} := (\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{\widetilde{U}_i}(\mathsf{tx}_i^{\mathsf{r}})))$.
- Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

2. Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$. Create the transaction $\mathsf{tx}_i^{\mathsf{p}} := \texttt{genPayTx}(\overline{\gamma_i}.\mathsf{st}, U_{i+1})$.
   - Generate signatures $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{p}})$ and set $\overline{\mathsf{tx}_i^{\mathsf{p}}} := (\mathsf{tx}_i^{\mathsf{p}}, (\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{p}})))$.
   - Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{p}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Lemma 17. *Let Σ be a EUF-CMA secure signature scheme. Then, the Pay phase of protocol Π GUC-emulates the Pay phase of functionality $\mathcal{F}_{Pay}$.*

Proof. We show that the simulator $\mathcal{S}$ presented above interacting with the Pay phase of $\mathcal{F}_{Pay}$ is indistinguishable for any environment $\mathcal{E}$ from an interaction with Π and a dummy adversary $\mathcal{A}$. A bit more formally, we show that the ensembles $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}}$ and $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ are indistinguishable for the environment $\mathcal{E}$.

In our description, we write $m[\tau]$ to denote that message m is observed at round τ. Moreover, we interact with other ideal functionalities, which in turn interact with either the environment $\mathcal{E}$ or other parties, who are possibly under adversarial control, by sending messages. These interactions can have an additional impact on publicly observable variables, i.e., the ledger $\mathcal{L}$. When sending a message m to an ideal functionality $\mathcal{F}$ in round τ, we denote the set of all by $\mathcal{E}$ observable actions triggered by this as a function $\mathsf{obsSet}(m, \mathcal{F}, \tau)$.

In the following, we analyze the different corruption cases. For each case, we first describe the view of the environment in Π and then the view of the environment as simulated by $\mathcal{S}$. For the Pay phase, we consider three different cases of the interaction between two users U_i and U_{i+1}. We match the sequences of this phase, that we use in the proof below, and where they are used in the ideal and real world in Table C.3. Note that for $U_i = U_0$ `SETUP` is performed initially, otherwise `CREATE_STATE`. We define the following messages.

- $m_0 := (\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}_i^{\mathsf{r}})$
- $m_1 := (\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_{i+1}}, \mathsf{R_i}, \mathsf{U_i}, \mathsf{U_{i+2}}, \alpha_\mathsf{i}, \mathsf{T})$

- $m_2 := (\mathtt{sid}, \mathtt{pid}, \mathtt{ACCEPT}, \overline{\gamma_{i+1}})$
- $m_3 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{\mathsf{U}_{i+1}}(\mathsf{tx}_i^r))$
- $m_4 := (\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i)$
- $m_5 := (\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i)$
- $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED})$ or, if sent by the receiver, $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx}^{\mathsf{er}}, \mathsf{T}, \alpha_i)$

Table C.3: Explanation of the sequence names used in Lemma 17 and where they can be found in the ideal functionality (IF), Protocol (Prot) or Simulator (Sim).

	Real World	Ideal World			Output	Description
		U_i honest, U_{i+1} corrupted	U_i honest, U_{i+1} honest	U_i corrupted, U_{i+1} honest		
SETUP	Prot.Pay.Setup 1-12	IF.Pay.Setup 1-7,9, Sim.Pay.a 1-5	IF.Pay.Setup 1-8	n/a	m_0	Does setup and contacts next user
CREATE_STATE	Prot.Pay.Open 6-8	n/a	IF.Pay.Open 12, 13 Sim.Pay.a 1-5	Sim.Pay.b 8-10	m_6, m_0	Upon m_5, sends message m_6 to $\mathcal{E}$. Then, ceates the objects to send in m_0 and sends it to U_{i+1} (or finalize).
CHECK_STATE	Prot.Pay.Open 1-4	n/a	IF.Pay.Open 1-8	Sim.Pay.b 1-4 IF.Check Sim.Pay.b 5-7 IF.Register	m_1, m_3	Checks if objects in m_0 are correct, sends m_1 to $\mathcal{E}$ and on m_2, sends m_3 to U_i
CHECK_SIG	Prot.Pay.Open 5	Sim.Pay.a 6	IF.Pay.Open 9-11	n/a	m_4	Checks if signature of tx_i^r is correct

1. U_i honest, U_{i+1} corrupted.

Real world: After U_i performs either SETUP or CREATE_STATE, it sends m_0 to U_{i+1} in the current round τ. The environment $\mathcal{E}$ controls $\mathcal{A}$ and therefore U_{i+1} and will see m_0 in round $\tau + 1$. Iff U_{i+1} replies with a correct message m_3 in $\tau + 2$, U_i will perform CHECK_SIG and call $\mathcal{F}_{Channel}$ with message m_4 in the same round. The ensemble is $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau + 1]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau + 2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either SETUP or simulator performs CREATE_STATE, the simulator sends m_0 to U_{i+1} in the current round τ. $\mathcal{E}$ will see m_0 in round $\tau+1$. Iff U_{i+1} replies with a correct message m_3 in $\tau+2$, the simulator will perform CHECK_SIG and call $\mathcal{F}_{Channel}$ with message m_4 in the same round. The ensemble is $\mathrm{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_0[\tau + 1]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau + 2)$

2. U_i honest, U_{i+1} honest.

Real world: After U_i performs either SETUP or CREATE_STATE, it sends m_0 to U_{i+1} in the current round τ. U_{i+1} performs CHECK_STATE and sends m_1 to $\mathcal{E}$ in round $\tau + 1$. Iff $\mathcal{E}$ replies with m_2, U_{i+1}, U_{i+1} replies with m_3. U_i receives this in round $\tau + 2$, performs CHECK_SIG and sends m_4 to $\mathcal{F}_{Channel}$. U_{i+1} expects the message m_5 in round $\tau + 2 + t_u$ and will then send m_6 to $\mathcal{E}$. Afterwards it continues with either CREATE_STATE or FINALIZE. The ensemble is $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_1[\tau + 1], m_6[\tau + 2 + t_u]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau + 2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either SETUP or is invoked by itself (in step Open.13) or by the simulator (in step process.6) in the current round τ, $\mathcal{F}_{Pay}$ perform the procedure Open. This behaves exactly like CREATE_STATE, CHECK_STATE and CHECK_SIG. However, since every object is created by $\mathcal{F}_{Pay}$, the checks are omitted. The procedure Open outputs the messages m_1 in round $\tau + 1$ and iff $\mathcal{E}$ replies with m_2, calls $\mathcal{F}_{Channel}$ with m_4 in $\tau + 2$. Finally, if m_5 is received in round $\tau + 2 + t_{\mathsf{u}}$, outputs m_6 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_1[\tau + 1], m_6[\tau + 2 + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_4, \mathcal{F}_{Channel}, \tau + 2)$

3. U_i **corrupted,** U_{i+1} **honest.**

Real world: After U_{i+1} receives the message m_0 from U_i, it performs CHECK_STATE and sends m_1 to $\mathcal{E}$ in the current round τ. Iff $\mathcal{E}$ replies with m_2, U_{i+1} sends m_3 to U_i. If U_{i+1} receives the message m_5 from $\mathcal{F}_{Channel}$ in round $\tau + 1 + t_{\mathsf{u}}$, it sends m_6 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_1[\tau], m_3[\tau + 1], m_6[\tau + 1 + t_{\mathsf{u}}]\}$

Ideal world: After the simulator receives m_0 from U_i, it performs CHECK_STATE together with $\mathcal{F}_{Pay}$ and $\mathcal{F}_{Pay}$ sends m_1 to $\mathcal{E}$. Iff $\mathcal{E}$ replies with m_2, $\mathcal{F}_{Pay}$ asks the simulator to send m_3 to U_i. All of this happens in the current round τ. If the simulator receives m_5 in round $\tau + 1 + t_{\mathsf{u}}$, it sends m_6 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_1[\tau], m_3[\tau + 1], m_6[\tau + 1 + t_{\mathsf{u}}]\}$

Note that we do not care about the case were both U_i and U_{i+1} are corrupted, because the environment is commuincating with itself, which is trivially the same in the ideal and the real world. We see that in these three cases, the execution ensembles of the ideal and the real world are identical, thereby proving Lemma 17.

□

Lemma 18. *Let Σ be a EUF-CMA secure signature scheme. Then, the Finalize phase of protocol Π GUC-emulates the Finalize phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles of the interaction between users U_n and U_0 for three different cases. We match the sequences and where they are used in the ideal and real world in Table C.4. We define the following messages.

- $m_7 := (\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{er}})$
- $m_8 := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{er}}})$

1. U_n **honest,** U_0 **corrupted.**

Table C.4: Explanation of the sequence names used in Lemma 18 and where they can be found.

	Real World	Ideal World			Output	Description
		U_n honest, U_0 corrupted	U_n honest, U_0 honest	U_n corrupted, U_0 honest		
FINALIZE	Prot.Pay.Open 8	IF.Pay.12 and Sim.Finalize.b or Sim.Pay.b 8	IF.Pay.12 and Sim.Finalize.b or Sim.Pay.b 8	n/a	m_7	Sends finalize message to U_0
CHECK_FINALIZE	Prot.Finalize 1-6	n/a	IF.Finalize 1,2,4 Sim.Finalize.a	Sim.Finalize.c IF.Finalize 1,3,4 Sim.Finalize.a	m_8	Checks if $\mathsf{tx}^{\mathsf{er}}$ is the same, if not, publishes it to ledger with m_8.

Real world: After performing FINALIZE in the current round τ, U_n sends m_7 to U_0. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_7[\tau]\}$

Ideal world: After either $\mathcal{F}_{Pay}$ or the simulator performs FINALIZE in the current round τ, the simulator sends m_7 to U_0. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_7[\tau]\}$

2. U_n honest, U_0 honest.

Real world: After performing FINALIZE in the current round τ, U_n sends m_7 to U_0. In the meantime, U_0 performs CHECK_FINALIZE and should it not receive a correct message m_7 in the correct round, will send m_8 to $\mathcal{G}_{Ledger}$ in round τ'. This will result in the sets of message The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

Ideal world: Either $\mathcal{F}_{Pay}$ or the simulator performs FINALIZE in the current round τ. In the meantime, $\mathcal{F}_{Pay}$ performs CHECK_FINALIZE and will, if the checks in FINALIZE failed or it was performed in a incorrect round τ', $\mathcal{F}_{Pay}$ will instruct the simulator to send m_8 to $\mathcal{G}_{Ledger}$ in rounds τ'. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

3. U_n corrupted, U_0 honest.

Real world: U_0 performs CHECK_FINALIZE and should it not receive a correct message m_7 in the correct round, will send m_8 to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

Ideal world: The simulator and $\mathcal{F}_{Pay}$ perform CHECK_FINALIZE and should the simulator not receive a correct message m_7 in the correct round, $\mathcal{F}_{Pay}$ will instruct the simulator to send m_8 to $\mathcal{G}_{Ledger}$ in round τ'.The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_8, \mathcal{G}_{Ledger}, \tau')$

□

Lemma 19. *Let Σ be a EUF-CMA secure signature scheme. Then, the Respond phase of protocol Π GUC-emulates the Respond phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles. This time only for the case were a user U_i is honest, however we distinguish between the case of revoke and force-pay. We match the sequences and where they are used in the ideal and real world in Table C.5. We define the following messages.

- $m_9 := (\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id})$
- $m_{10} := (\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}})$
- $m_{11} := (\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED})$
- $m_{12} := (\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_{i-1}^{\mathsf{p}}})$
- $m_{13} := (\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}PAY})$

Table C.5: Explanation of the sequence names used in Lemma 19 and where they can be found.

	Real World	Ideal World	Output	Description
		U_i honest		
RESPOND	Prot.Respond	IF.Respond	n/a	Checks every round if response in order.
REVOKE	Prot.Respond.1	IF.Respond.Revoke Sim.Respond.1	m_9, m_{10}, m_{11}	Carries out the revokation.
FORCE_PAY	Prot.Respond.2	IF.Respond.Revoke Sim.Respond.2	m_{12}, m_{13}	Carries out the force-pay.

U_i honest, revoke.

Real world: In every round τ, U_i performs RESPOND, which provides a decision on whether or not to do the following. If yes, U_i performs REVOKE, which results in message m_9 to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_9 is closed, U_i sends m_{10} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{10} appears on $\mathcal{L}$ in $\tau + t_{\mathsf{c}} + 2\Delta$, U_i sends m_{11} to $\mathcal{E}$. The ensemble is $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{11}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_9, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{10}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs RESPOND, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform REVOKE, which results in the message m_9 to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_9 is closed, the simulator sends m_{10} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{10} appears on $\mathcal{L}$, $\mathcal{F}_{Pay}$ sends m_{11} to $\mathcal{E}$. The ensemble is $\mathrm{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{11}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_9, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{10}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

U_i honest, force-pay.

Real world: In every round τ, U_i performs RESPOND, which provides a decision on whether or not to do the following. If yes, U_i performs FORCE_PAY, which results in the messages m_{12} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{12} appears on $\mathcal{L}$, the message m_{13} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{12}, \mathcal{G}_{Ledger}, \tau)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs RESPOND, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform FORCE_PAY, which results in the messages m_{12} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{12} appears on $\mathcal{L}$, the message m_{13} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{12}, \mathcal{G}_{Ledger}, \tau)$

□

Theorem 10. *(formal) Let Σ be a EUF-CMA secure signature scheme. Then, for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

This theorem follows directly from Lemma 17, 18 and Lemma 19.

C.11 Discussion on security and privacy goals

So far, in Section 4.4.1 we have informally stated what are our security and privacy goals in this work. Additionally, in Appendix C.10.4 we have described the ideal functionality $\mathcal{F}_{Pay}$ that formally defines the security and privacy guarantees achieved by Blitz. In this section, we aim to show how $\mathcal{F}_{Pay}$ indeed has the security and privacy goals that we intuitively want to achieve. For that, we first formalize each intuitive security and privacy goal into a cryptographic game and then show that $\mathcal{F}_{Pay}$ fulfills such definition.

C.11.1 Assumptions

For the theorems in this section, we have the following assumptions: (i) we assume that stealth addresses achieve unlinkability and (ii) we assume that the routing scheme we use (i.e., Sphinx extended with a per-hop payload) is a secure onion routing process.

Unlinkability of stealth addresses. Consider the following game. The challenger computes two pair of stealth addresses (A_0, B_0) and (A_1, B_1). Moreover, the challenger picks a bit b and computes $P_b, R_b \leftarrow \mathsf{GenPk}(A_b, B_b)$. Finally, the challenger sends the tuples (A_0, B_0), (A_1, B_1) and P_b, R_b to the adversary.

Additionally, the adversary has access to an oracle that upon being queried, returns P_b^*, R_b^* to the adversary.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 10 (Unlinkability of Stealth Addresses)**.** We say that a stealth addresses scheme achieves unlinkability if for all PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Secure onion routing process. We say that an onion routing process is secure, if it realizes the ideal functionality defined in [CL05]. Sphinx [DG09], for instance, is a realization of this. We use it in Blitz, extended with a per-hop payload (see also Section 4.4.2).

C.11.2 Balance security

Given a path $\mathsf{channelList} := \gamma_1, \ldots, \gamma_n$ and given a user U such that $\gamma_i.\mathsf{right} = U$ and $\gamma_{i+1}.\mathsf{left} = U$, we say that the balance of U in the path is $\mathbf{PathBalance}(U) := \gamma_i.\mathsf{balance}(U) + \gamma_{i+1}.\mathsf{balance}(U)$. Intuitively then, we say that a payment protocol achieves *balance security* if the $\mathbf{PathBalance}(U)$ for each honest user U does not decrease.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α, and a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets $\mathtt{sid}$ and $\mathtt{pid}$ to two random identifiers. Then, the challenger simulates a payment from the setup phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. The challenger runs the Pay phase. Every time that a corrupted user U_i needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer, thereby giving the attacker the opportunity to stop payments and trigger refunds or let them be successful.

We say that the adversary wins the game if there exists an honest intermediate user U, such that $\mathbf{PathBalance}(U)$ is lower after the payment operation.

Definition 11 (Balance security)**.** We say that a payment protocol achieves balance security if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 11 (Blitz achieves balance security)**.** *Blitz payments achieve balance security as defined in Definition 11.*

Proof. Assume that an adversary exists, can win the balance privacy game. This means, that after the balance security game, there exists an honest intermediate user U, such $\mathbf{PathBalance}(U)$ is lower after the payment. An intermediary U has coins locked up in its right channel when $\mathcal{F}_{Pay}$ (if right neighbor is honest) or the simulator (if right neighbor dishonest) updates this right channel to its new state. However, both $\mathcal{F}_{Pay}$ and the simulator do this only, after successfully updating also their left channel using the same $\mathsf{tx}^{\mathsf{er}}$ to fund the refund transactions in both channels.

Assume now that U has less channel balance after the payment. This would imply, that U lost its fund in the right channel without gaining any in the left. Consider two cases: (i) The left neighbor refunded in time, which implies that $\mathsf{tx}^{\mathsf{er}}$ was posted in time, which triggers also the refund in $\mathcal{F}_{Pay}$ of U in its right channel and no balance is lost. (ii) The right neighbor claimed the collateral of U's right channel. Since for an honest U, $\mathcal{F}_{Pay}$ would have automatically refunded before T if possible, this means that also in U's left channel, no refund occurred. Therefore, U can claim the money put by its left neighbor and will not lose balance. This leads to the conclusion, that no such honest U exists with a lower channel balance, or if its the sender, a lower channel balance and an unsuccessful payment. □

C.11.3 Sender privacy

Intuitively, we say that a payment protocol achieves *sender privacy* if an adversary controlling an intermediary node cannot distinguish the case where the sender is its left neighbor in the path from the case where the sender is separated for one (or more) intermediaries.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the number of intermediary nodes between the sender and the adversary. In particular, the path $\mathsf{channelList}_0$ is formed by U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users U_0, U_1, U^*, U_2. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α as well as a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}_b$. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha + \mathsf{fee}, T)$ to the challenger. The challenger sets `sid` and `pid` to two random identifiers. Moreover, the challenger picks a bit b at random and simulates setup and open of the Pay phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 12 (Sender privacy)**.** We say that a payment protocol achieves sender privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Theorem 12 (Blitz achieves sender privacy)**.** *Blitz payments achieve sender privacy as defined in Definition 12.*

Proof. The message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\epsilon_i})$ that $\mathcal{F}_{Pay}$ sends to the simulator in the Open phase, is leaked to the adversary. By looking at $\overline{\gamma_i}$ and

opening onion_{i+1}, U^* knows its neighbors U_1 and U_2. We know that U^* cannot learn any additional information about the path from T and $\overline{\gamma_i}$. Since the amount to be sent was increased fee for the path $\mathsf{channelList}_1$, the amount α_i for U_i is identical for both cases. This leaves $\mathsf{tx}^{\mathsf{er}}$, rList and onion_{i+1}. Let us assume, that there exists an adversary that can break sender privacy. There are two possible cases.

1. The adversary finds out by looking at $\mathsf{tx}^{\mathsf{er}}$ and rList: We defined that the output, that serves as input for $\mathsf{tx}^{\mathsf{er}}$, has never been used and is unlinkable to the sender and check this in `checkTxIn`. Looking at the outputs of $\mathsf{tx}^{\mathsf{er}}$, the adversary knows to whom all but one output belongs. Since our adversary breaks the sender privacy, it needs to be able to reconstruct, to whom this final output of $\mathsf{tx}^{\mathsf{er}}$ belongs observing rList. This contradicts our assumption of unlinkable stealth addresses.

2. The adversary finds out by looking at onion_{i+1}: The adversary controlling U^* is able to open onion_{i+1} revealing U_2, a message m and onion_{i+2}. Since our adversary breaks the sender privacy, he has to be able to open onion_{i+2} to reveal if U_2 is the receiver or not, thereby learning who is the sender. This contradicts our assumption of secure anonymous communication networks.

These two cases lead to the conclusion, that a PPT adversary that can win the sender privacy game with a probability non-negligibly better than $1/2$, can also break our assumptions of unlinkability of stealth addresses or secure anonymous communication networks. Note that both receiver privacy and its proof are analogous to the sender privacy. □

C.11.4 Path privacy

Intuitively, we say that a payment protocol achieves *path privacy* if an adversary controlling an intermediary node does not know what other nodes are part of the path other than its own neighbors.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the nodes other than the adversary neighbors. In particular, the path $\mathsf{channelList}_0$ is formed by U_0, U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users $U'_0, U_1, U^*, U_2, U'_3$. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Further note that we force that in both paths, the adversary has the same neighbors as otherwise there exists a trivial distinguishability attack based on what neighbors are used in each case.

Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a payment amount α as well as a timeout T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $n \cdot \epsilon$ coins. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Moreover, the challenger picks a bit b at random and simulates the setup and open phases on input

$(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 13 (Path privacy)**.** We say that a payment protocol achieves path privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Theorem 13 (Blitz achieves path privacy)**.** *Blitz payments achieve path privacy as defined in Definition 13.*

Proof. As this proof is analogous to the proof for sender privacy, we only sketch it here. Again, the simulator leaks the same message $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{er}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_i}, \theta_{\epsilon_i})$ to the adversary. Again, the adversary can find out the correct bit b by looking at (i) $\mathsf{tx}^{\mathsf{er}}$ and rList or (ii) at onion_{i+1}. If there exists an adversary that breaks the path privacy of Blitz, then it also can be used to break (i) unlinkability of stealth addresses or (ii) secure anonymous communication networks. □

APPENDIX D

Appendix to Chapter 5

D.1 Stealth addresses

The stealth addresses scheme allows us to derive one-time and fresh public keys in a digital signature scheme for a specific user. Here, we briefly describe a basic dual-key stealth addresses protocol (DKSAP). Assume that G is a base point of an elliptic curve, in which the difficulty of the elliptic curve discrete logarithm problem (ECDLP) [Kob87] holds. Moreover, assume that there is a user (say Alice) with two pairs of private/public keys $(a, A), (b, B)$ such that $A = a \cdot G$ and $B = b \cdot G$. We want to derive fresh public keys for Alice. A DKSAP is a tuple of two algorithms DKSAP := $(\mathsf{GenPk}, \mathsf{GenSk})$ defined as follows.

- $(P, R) \leftarrow \mathsf{GenPk}(A, B)$: A PPT algorithm takes two Alice's public keys A, B as inputs and returns a fresh public key for Alice P along with an additional value R, which is required for deriving the secret key for P. For that, a random $r \leftarrow^{\$} [0, l-1]$ is sampled uniformly, where l is the prime order of the underlying elliptic curve. Then, P is computed as $P := \mathcal{H}(r \cdot A) \cdot G + B$, $\mathcal{H}$ is a hash function modeled as a random oracle. Moreover, R is computed as $R := r \cdot G$.

- $p \leftarrow \mathsf{GenSk}(a, b, P, R)$: A DPT algorithm takes two Alice's secret keys a, b and P, R generated by GenPk algorithm as inputs and returns the secret key corresponding to P. For that p is computed as $p := \mathcal{H}(a \cdot R) + b$.

Correctness of algorithms follows directly: $p \cdot G = (\mathcal{H}(a \cdot R) + b) \cdot G = \mathcal{H}(a \cdot r \cdot G) \cdot G + b \cdot G = \mathcal{H}(r \cdot A) \cdot G + B = P$. In [VS18] it is argued that the new address P is unlikable for a spectator, even when observing R.

D.2 UC modeling

We now formalize our construction in the global UC framework (GUC) [CDPW07], which is an extension of the standard UC framework [Can01] that allows for a global setup. We use this global step for modeling the ledger. Through this section, first, we provide some preliminaries. Then, we define an ideal functionality for the multi-channel updates protocol. Our model follows closely the model in [AEE+21, AMSKM21, AMSKM23].

D.2.1 Preliminaries, communication model and threat model

In the real world, a protocol Π is executed by a set of parties $\mathcal{P}$ and in the presence of an adversary $\mathcal{A}$. A security parameter $\lambda \in \mathbb{N}$ and an auxiliary input $z \in \{0,1\}^*$ are given to the adversary as inputs. We consider a static corruption model, which means that $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the protocol execution. $\mathcal{A}$ controls corrupted parties and learns their internal states. All parties in $\mathcal{P}$ and $\mathcal{A}$ take their input from a special entity called environment $\mathcal{E}$, which represents everything external to the protocol. This entity observes all output messages from participants. We assume that the communication network is synchronous, and the protocol execution takes place in rounds. The global ideal functionality $\mathcal{G}_{clock}$ [KMTZ13] represents a global clock that proceeds to the next round if all honest parties indicate that they are ready to do so. Every entity always knows the current round. Communications between parties in $\mathcal{P}$ are through authenticated channels with guaranteed delivery after exactly one round. If a party P sends a message to party Q in round t, then Q receives that message in the beginning of round $t+1$ and knows that P has sent the message. We model authenticated channels by an ideal functionality $\mathcal{F}_{GDC}$ [DEF+19b]. The adversary can read and reorder the messages sent in the same round, but can not modify or delay messages. Communications involving $\mathcal{A}$, $\mathcal{E}$, or the simulator $\mathcal{S}$ and every computation that a party executes locally take zero rounds.

D.2.2 Ledger and channels

We model a UTXO-based blockchain in the ideal functionality $\mathcal{G}_{Ledger}$. We denote the blockchain delay as Δ, and the blockchain's signature scheme by Σ. $\mathcal{G}_{Ledger}$ communicates with a fixed set of parties $\mathcal{P}$.

Initially, the environment $\mathcal{E}$ chooses a key pair (sk_P, pk_P) for each $P \in \mathcal{P}$ and registers it to the ledger by sending
$(\texttt{sid}, \texttt{register}, pk_P)$ to $\mathcal{G}_{Ledger}$. Also, $\mathcal{E}$ sets the initial state of $\mathcal{L}$, which is a publicly accessible set of all published transactions. A party $P \in \mathcal{P}$ can post a transaction $\overline{\mathsf{tx}}$ via message $(\texttt{sid}, \texttt{POST}, \overline{\mathsf{tx}})$ to $\mathcal{G}_{Ledger}$. The transaction will be added to the ledger after at most Δ rounds if it is valid. The exact number of delay rounds is chosen by the adversary. In this work, we consider a simplified model for the underlying blockchain and assume that the set of users is fixed instead of allowing them to join or leave dynamically. For a more precise model, we refer the reader to works [BMTZ17]. We define an ideal functionality $\mathcal{F}_{Channel}$ [AME+21], which is built on top of $\mathcal{G}_{Ledger}$ and provides *open*,

update, and *close* procedures related to payment channels. We assume that closing a channel takes at most t_c rounds and updating a channel takes at most t_u rounds. For simplicity, we assume that channels involved in the multi-channel updates protocol have already been registered and opened with the ledger functionality.

The complete API of $\mathcal{F}_{Channel}$ and $\mathcal{G}_{Ledger}$ are shown below. We hide the calls to $\mathcal{G}_{clock}$ and $\mathcal{F}_{GDC}$ in our notation. Instead of explicitly calling these functionalities, we write $\mathsf{msg} \overset{t}{\hookrightarrow} X$ to denote sending message msg to party X in round t and also $\mathsf{msg} \overset{t}{\hookleftarrow} X$ to denote receiving message msg from party X in round t.

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma)$ [AEE+21, AMSKM21]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accepted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions .

Parameters:

- Δ: upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$
- Σ: a digital signature scheme

API:

Messages from $\mathcal{E}$ via a dummy user $P \in \mathcal{P}$:

- $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_P) \overset{\tau}{\hookleftarrow} P$:
  This function adds an entry (pk_P, P) to PKI consisting of the public key pk_P and the user P, if it does not already exist.
- $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}}) \overset{\tau}{\hookleftarrow} P$:
  This function checks if $\overline{\mathsf{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

Interface of $\mathcal{F}_{Channel}(T, k)$ [AEE+21, AMSKM21]

Parameters:

- T: upper bound on the maximum number of consecutive off-chain communication rounds between channel users
- k: number of ways the channel state can be published on the ledger

API:

Messages from $\mathcal{E}$ via a dummy user P:

- $(\mathtt{sid}, \mathtt{CREATE}, \overline{\gamma}, \mathsf{tid}_P) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the attribute tuple $(\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users}, \overline{\gamma}.\mathsf{cash}, \overline{\gamma}.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\overline{\gamma}.\mathsf{users} \subset \mathcal{P}$ are the users of the channel (and $P \in \overline{\gamma}.\mathsf{users}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is the total money in the channel and $\overline{\gamma}.\mathsf{st}$ is the initial state of the channel. tid_P defines P's input for the funding transaction of the channel. When invoked, this function asks $\overline{\gamma}.\mathsf{otherParty}$ to create a new channel.

- $(\texttt{sid}, \texttt{UPDATE}, \mathsf{id}, \theta) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the channel where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$. When invoked by $P \in \overline{\gamma}.\mathsf{users}$ and both parties agree, the channel $\overline{\gamma}$ (if it exists) is updated to the new state θ. If the parties disagree or at least one party is dishonest, the update can fail or the channel can be forcefully closed to either the old or the new state. Regardless of the outcome, we say that t_{u} is the upper bound that an update takes. In the successful case, $(\texttt{sid}, \texttt{UPDATED}, \mathsf{id}, \theta) \overset{\leq \tau + t_{\mathsf{u}}}{\hookleftarrow} \overline{\gamma}.\mathsf{users}$ is output.
- $(\texttt{sid}, \texttt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookleftarrow} P$:
  Will close the channel $\overline{\gamma}$, where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$, either peacefully or forcefully. After at most t_{c} in round $\leq \tau + t_{\mathsf{c}}$, a transaction tx with the current state $\overline{\gamma}.\mathsf{st}$ as output ($\mathsf{tx}.\mathsf{output} := \overline{\gamma}.\mathsf{st}$) appears on $\mathcal{L}$ (the public ledger of $\mathcal{G}_{Ledger}$).

D.2.3 The UC-security definition

Closely following [AMSKM21, AMSKM23], we define Π as a *hybrid* protocol that accesses to ideal functionality $\mathcal{F}_{prelim}$ consisting of $\mathcal{F}_{GDC}$, $\mathcal{G}_{Ledger}$, $\mathcal{F}_{Channel}$, and $\mathcal{G}_{clock}$. In the beginning, the environment $\mathcal{E}$ supplies inputs to the parties in $\mathcal{P}$ and the adversary $\mathcal{A}$ with a security parameter λ and auxiliary input z. We denote the output that $\mathcal{E}$ observes as the ensemble $\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. $\Phi_{\mathcal{F}_{update}}$ denotes the ideal protocol of the ideal functionality $\mathcal{F}_{update}$, where the dummy users simply forward their input to $\mathcal{F}_{update}$. With access to functionalities $\mathcal{F}_{prelim}$, we denote the output of this idealized protocol as $\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Phi_{\mathcal{F}_{update}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

If a protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{update}$, then any attack that is possible on the real world protocol Π can be carried out against the ideal protocol $\Phi_{\mathcal{F}_{update}}$ and vice versa.

Definition 14. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{update}$, w.r.t. $\mathcal{F}_{prelim}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any $z \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$, we have

$$\texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z) \approx_c \texttt{EXEC}^{\mathcal{F}_{prelim}}_{\Phi_{\mathcal{F}_{update}},\mathcal{S},\mathcal{E}}(\lambda, z) \tag{D.1}$$

where $\approx_c$ denotes computational indistinguishability.

D.2.4 Ideal functionality

Here, we define our ideal functionality $\mathcal{F}_{update}$. This functionality can output an ERROR message, e.g., when a transaction does not appear on the ledger as it should. When $\mathcal{F}_{update}$ outputs ERROR, any guarantees are lost. Hence, we are only interested in protocols that realize $\mathcal{F}_{update}$ and never output an ERROR. The subprocedures used in $\mathcal{F}_{update}$, Π, and $\mathcal{S}$ follow the same logic as the macros defined in Section 5.4.2.

Note that in $\mathcal{F}_{update}$ and Π, for better readability, we use the set $\mathcal{P}$ to store all parties, the set $\mathcal{S}$ to store all senders, and the set $\mathcal{R}$ to store all receivers. We know that two

different channels may have a common user. Thus, for handling duplicated identifiers in the aforementioned sets, we implicitly assign different identifiers for users of different channels. Consequently, the size of each set is equal to the number of channels.

Ideal Functionality $\mathcal{F}_{update}(\Delta, T)$

Parameters:

Δ : Upper bound on the time it takes a transaction to appear on $\mathcal{L}$.

T : Upper bound on the time expected for successful payments.

Local variables:

idSet : A set of tuples containing pairs of ids and channels $(\texttt{pid}, \gamma_i)$ to avoid duplicated channels.

Γ : A set of tuples $(\texttt{pid}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}, \theta_{i,j}\}_{j\in[1,n]})$ that for each payment id pid and channel $\overline{\gamma}_i$, store the state transaction $\mathsf{tx}_i^{\mathsf{state}}$, refund transaction $\mathsf{tx}_i^{\mathsf{r}}$ and a set of tuples for payment transactions $(\mathsf{tx}_{i,j}^{\mathsf{p}}, \theta_{i,j})$ where $\theta_{i,j}$ is the output of $\mathsf{tx}_j^{\mathsf{ep}}$ used in $\mathsf{tx}_{i,j}^{\mathsf{p}}$.

Ψ : A map, storing for a given pid a copy of all $\mathsf{tx}^{\mathsf{ep}}$ in a set $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$.

t_{u} : Time required to perform a ledger channel update honestly.

t_c : Time it takes at most to close a channel.

Start (executed in the beginning in round τ_{s})

Send $(\texttt{sid}, \texttt{start}) \overset{\tau_{\mathsf{s}}}{\hookrightarrow} \mathcal{S}$ and upon $(\texttt{sid}, \texttt{start-ok}, t_{\mathsf{u}}, t_{\mathsf{c}}) \overset{\tau_{\mathsf{s}}}{\hookleftarrow} \mathcal{S}$ set t_{u} and t_{c} accordingly.

Initialization

Let τ be the current round, and $\mathcal{S}$, $\mathcal{R}$, and $\mathcal{P}$ be initially empty sets.

1. If $(\texttt{sid}, \texttt{pid}, \texttt{CHANNELS-SET}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow} \mathsf{dealer}$ where the dealer is honest, do the following.
   a) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-init}, \{\gamma_j\}_{j\in[1,n]}, \mathsf{dealer}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.
   b) For all honest $P_i \in \{\gamma_i.\mathsf{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$, send $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECK}, \{\gamma_j\}_{j\in[1,n]}) \overset{\tau+1}{\longrightarrow} P_i$.

2. Upon each message $(\texttt{sid}, \texttt{pid}, \texttt{send-check}, \{\gamma_i\}_{i\in[1,n]}, P_i) \xleftarrow{\tau+1} \mathcal{S}$, send $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECK}, \{\gamma_j\}_{j\in[1,n]}) \xrightarrow{\tau+1} P_i$.
3. Upon $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xleftarrow{\tau+1} P_i$ for each honest P_i, do following.
   a) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-init-ok}, \{\gamma_j\}_{j\in[1,n]}, P_i) \xrightarrow{\tau+1} \mathcal{S}$.
   b) If this is the first INIT-CHECKED message from an honest party, for each γ_i the tuple $(\texttt{pid}, \gamma_i) \notin \mathsf{idSet}$, set $\mathsf{idSet} = \mathsf{idSet} \cup \{(\texttt{pid}, \gamma_i)\}$, add $\gamma_i.\mathsf{sender}$ to $\mathcal{S}$ and $\mathcal{P}$, and add $\gamma_i.\mathsf{receiver}$ to $\mathcal{R}$ and $\mathcal{P}$.
4. If there is an honest $P_i \in \mathcal{P}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xleftarrow{\tau+1} P_i$ is not received, go idle.
5. If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{P}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{init-acc}, P_i, P_j) \xleftarrow{\tau+2} \mathcal{S}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
6. Go to the *Pre-Setup* phase, and pass the set of channels with the receiver in $\mathcal{P}$ to the next phase.

Pre-Setup

Let τ be the current round.

1. For each channels γ_i do following.
   a) Let $\mathsf{tx}_i^{\mathsf{in}} := \texttt{GenTxIn}(\gamma_i.\mathsf{receiver}, \{\gamma_k\}_{k\in[1,n]})$.
   b) Let $\mathsf{tx}_i^{\mathsf{ep}} := \texttt{GenTxEp}(\{\gamma_k\}_{k\in[1,n]}, \mathsf{tx}_i^{\mathsf{in}})$, and add $\mathsf{tx}_i^{\mathsf{ep}}$ to $\Psi(\texttt{pid})$
   c) If $\gamma_i.\mathsf{receiver}$ is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{presetup-req}, \gamma_i, \mathsf{tx}_i^{\mathsf{ep}}) \xhookrightarrow{\tau} \mathcal{S}$.
   d) Else if $\gamma_i.\mathsf{receiver}$ is honest, for all corrupted $P_j \in \mathcal{P}$ send $(\texttt{sid}, \texttt{pid}, \texttt{send-presetup}, \mathsf{tx}_i^{\mathsf{ep}}, \gamma_i.\mathsf{receiver}, P_j) \xhookrightarrow{\tau} \mathcal{S}$.
2. If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{presetup-acc}, P_i, P_j) \xleftarrow{\tau+1} \mathcal{S}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
3. Go to the *Setup* phase, and pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

Setup

Let τ be the current round.

1. For each channel γ_i if both $\gamma_i.\mathsf{sender}$ and $\gamma_i.\mathsf{receiver}$ are honest, do the following.
   a) If $\gamma_i.\mathsf{sender} \in \mathcal{P}$, $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xhookrightarrow{\tau} \gamma_i.\mathsf{sender}$.

b) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftarrow{\tau} \gamma_i.\mathsf{sender}$, continue. Otherwise skip the steps (c) to (g).

c) Let $\mathsf{tx}_i^{\mathsf{state}} := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, and $\mathsf{tx}_i^{\mathsf{r}} := \texttt{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{sender})$.

d) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to $\gamma_i.\mathsf{receiver}$, then create $\mathsf{tx}_{i,j}^{\mathsf{p}} = \texttt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$.

e) If $\gamma_i.\mathsf{receiver} \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xrightarrow{\tau+1} \gamma_i.\mathsf{receiver}$.

f) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftarrow{\tau+1} \gamma_i.\mathsf{receiver}$, continue. Otherwise skip the step (g).

g) For all corrupted $P_j \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup-ok}, \gamma_i.\mathsf{receiver}, P_j) \xrightarrow{\tau+1} \mathcal{S}$.

2. Else If $\gamma_i.\mathsf{sender}$ is corrupted and $\gamma_i.\mathsf{receiver}$ is honest, do the following.

a) If $(\texttt{sid}, \texttt{pid}, \texttt{setup-acc}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}\}_{j\in[1,n]}) \xleftarrow{\tau+1} \mathcal{S}$, set $\alpha_i := \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}[0].\mathsf{cash}$. Otherwise, skip the steps (b) to (d).

b) If $\gamma_i.\mathsf{receiver} \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xrightarrow{\tau+1} \gamma_i.\mathsf{receiver}$.

c) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftarrow{\tau+1} \gamma_i.\mathsf{receiver}$ with a same α_i as the step(b) and $\mathsf{tx}_i^{\mathsf{state}} = \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, continue. Otherwise skip the step (e).

d) For all corrupted $P_j \in \mathcal{P}$, send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup-ok}, P_i, P_j) \xrightarrow{\tau+1} \mathcal{S}$.

3. Else If $\gamma_i.\mathsf{sender}$ is honest and $\gamma_i.\mathsf{receiver}$ is corrupted, do the following.

a) If $\gamma_i.\mathsf{sender} \in \mathcal{P}$, $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xhookrightarrow{\tau} \gamma_i.\mathsf{sender}$.

b) Upon $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftarrow{\tau} \gamma_i.\mathsf{sender}$, continue. Otherwise skip the steps (c) to (e).

c) Let $\mathsf{tx}_i^{\mathsf{state}} := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, and $\mathsf{tx}_i^{\mathsf{r}} := \texttt{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{sender})$.

d) For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to $\gamma_i.\mathsf{receiver}$, then create $\mathsf{tx}_{i,j}^{\mathsf{p}} = \texttt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$.

e) Send $(\texttt{sid}, \texttt{pid}, \texttt{send-setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}) \xrightarrow{\tau+1} \mathcal{S}$.

4. If there is an honest receiver $P_i \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xleftarrow{\tau+1} P_i$ is not received, go idle.

5. If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\texttt{sid}, \texttt{pid}, \texttt{setup-finalized}, P_i, P_j) \xleftarrow{\tau+1} \mathcal{S}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.

6. Go to the *Confirmation* phase. Pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

Confirmation

- Let τ be the current round.

1. For each honest sender $\gamma_i.\mathsf{sender} \in \mathcal{S}$, do the following.
   a) Send $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$.
   b) If not $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \overset{\tau+t_u}{\hookleftarrow} \mathcal{F}_{Channel}$, skip the step (c).
   c) For each corrupted $\gamma_j.\mathsf{receiver} \in \mathcal{R}$, send
   $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}sig}, \gamma_i.\mathsf{sender}, \gamma_j.\mathsf{receiver}, \mathsf{tx}_j^{\mathsf{ep}}) \overset{\tau+t_u}{\hookrightarrow} \mathcal{S}$.
2. For each honest receiver $\gamma_i.\mathsf{receiver} \in \mathcal{R}$, if
   (i) $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}acc}, \gamma_i.\mathsf{receiver}, \gamma_j.\mathsf{sender}) \overset{\tau+t_u+1}{\hookleftarrow} \mathcal{S}$ is received for all corrupted $\gamma_j.\mathsf{sender} \in \mathcal{S}$, and
   (ii) $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \overset{\tau+t_u}{\hookleftarrow} \mathcal{F}_{Channel}$ on behalf of $\gamma_i.\mathsf{receiver}$, do the following.
   a) Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED}, \overline{\gamma}_i) \overset{\tau+t_u+1}{\hookrightarrow} \gamma_i.\mathsf{receiver}$
   b) For all corrupted $P_j \in \mathcal{P}$,
   $(\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}confirmation\text{-}ok}, \gamma_i.\mathsf{receiver}, P_j) \overset{\tau+t_u}{\hookrightarrow} \mathcal{S}$.
3. If there is an honest receiver $\gamma_i.\mathsf{receiver}$, where
   $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}acc}, \gamma_i.\mathsf{receiver}, \gamma_j.\mathsf{sender}) \overset{\tau+t_u+1}{\hookleftarrow} \mathcal{S}$ is not received for at least one corrupted $\gamma_j.\mathsf{sender} \in \mathcal{S}$, or
   $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \overset{\tau+t_u}{\hookleftarrow} \mathcal{F}_{Channel}$ on behalf of $\gamma_i.\mathsf{receiver}$, go idle.
4. If there is an honest $P_i \in \mathcal{P}$ and a corrupted $P_j \in \mathcal{R}$, where the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmation\text{-}finalized}, P_i, P_j) \overset{\tau+t_u+1}{\hookleftarrow} \mathcal{S}$ is not received, remove P_i from $\mathcal{P}$ and $\mathcal{S}$ or $\mathcal{R}$.
5. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{agg\text{-}sig}, \{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}, \mathcal{S}) \overset{\tau+t_u+1}{\hookrightarrow} \mathcal{S}$.
6. Go to the *Finalizing* phase. Pass the set of channels with at least one user in $\mathcal{P}$ to the next phase.

Finalizing

- Let τ be the current round.

1. For each channel γ_i, let $\mathsf{tx}_i^{\mathsf{trans}} := \mathtt{GenTrans}(\alpha_i, \overline{\gamma}_i)$.
2. For each honest sender $\gamma_i.\mathsf{sender}$, send
   $(\mathtt{ssid}_C, \mathtt{UPDATE}, \gamma_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$.
3. For each channels γ_i, If $\gamma_i.\mathsf{receiver}$ is honest, do the following.

a) If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output}) \xleftarrow{\tau+t_u} \mathcal{F}_{Channel}$, $(\texttt{sid}, \texttt{pid}, \texttt{post-txep}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{ep}}) \xrightarrow{\tau+t_u} \mathcal{S}$.

b) Send $(\texttt{sid}, \texttt{pid}, \texttt{FINALIZED}, \overline{\gamma}_i) \xrightarrow{\tau+t_u} \gamma_i.\mathsf{receiver}$.

Respond (executed at the end of every round)

Let t be the starting round. For every element $(\texttt{pid}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}, \theta_{i,j}\}_{j\in[1,n]}) \in \Gamma$, if $\overline{\gamma}_i.\mathsf{st} = \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}$, and one $\mathsf{tx}_j^{\mathsf{ep}} \in \Psi(\texttt{pid})$ is on $\mathcal{L}$, do the Pay step as follows.

Pay: If $\gamma_i.\mathsf{receiver}$ is honest and $t < T - t_c - 2\Delta$ do the following.

1. $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$
2. At time $t + t_c$, if a transaction tx with $\mathsf{tx}.\mathsf{output} = \overline{\gamma}_i.\mathsf{st}$ appears on $\mathcal{L}$, Wait for Δ rounds and send $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma}_i, \mathsf{tx}_{i,j}^{\mathsf{p}}) \xrightarrow{t'<T-\Delta} \mathcal{S}$.
3. At time $t'' < T$, if a transaction tx' appears on $\mathcal{L}$ with $\mathsf{tx}'.\mathsf{input} = [\theta_{i,j}, \mathsf{tx}.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = [(\mathsf{tx}.\mathsf{output}[0].\mathsf{cash} + \theta_{i,j}.\mathsf{cash}, \mathsf{OneSig}(\gamma_i.\mathsf{receiver}))]$, send $(\texttt{sid}, \texttt{pid}, \texttt{PAID}) \overset{t''}{\hookrightarrow} \gamma_i.\mathsf{receiver}$. Otherwise return ERROR to all parties.

Force-Refund: Else, if a transaction tx with $\mathsf{tx}.\mathsf{output} = \overline{\gamma}_i.\mathsf{st}$ is on-chain and $\mathsf{tx}.\mathsf{output}[0]$ is unspent, $t \geq T$, and $\gamma_i.\mathsf{sender}$ is honest, do the following.

1. Send $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{r}}) \overset{t}{\hookrightarrow} \mathcal{S}$
2. If transaction tx' with $\mathsf{tx}'.\mathsf{input} = [\mathsf{tx}.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = (\mathsf{tx}.\mathsf{output}[0].\mathsf{cash}, \mathsf{OneSig}(\gamma_i.\mathsf{sender}))$ appears on $\mathcal{L}$ in round $t_1 < t+\Delta$, send $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-REFUND}) \overset{t_1}{\hookrightarrow} \gamma_i.\mathsf{sender}$. Otherwise, return ERROR to all parties.

D.2.5 Protocol

In this section, we present the formal protocol Π. The protocol is similar to what is presented in Figure 5.5, but extended with payment ids and UC formalism. We add the environment $\mathcal{E}$ and model communication in rounds. The protocol is divided into six phases. In *Initialization*, a user dealer receives the ongoing updates from $\mathcal{E}$ and sends them to every user to check whether all participants agree with that. In *Pre-Setup*, each receiver generates $\mathsf{tx}^{\mathsf{ep}}$ and sends it to all parties. In *Setup*, senders generate and send $\mathsf{tx}^{\mathsf{state}}$, tx^{p}, and tx^{r} to their neighbors. Receivers verify the messages and inform all parties when everything is OK. In *Confirmation*, senders update their channels and then send their signature to each $\mathsf{tx}^{\mathsf{ep}}$ to the corresponding receivers. When a receiver gets all signatures, sends an endorsement to all parties. In *Finalizing*, the senders after receiving all endorsements update their channel to the final state. If a receiver does not get UPDATED from $\mathcal{F}_{Channel}$, puts $\mathsf{tx}^{\mathsf{ep}}$ on-chain. In *Respond* users will react to $\mathsf{tx}^{\mathsf{ep}}$ being published and, either force payments or refunds.

Protocol Π

Local variables:

pidSet : A set storing every payment id `pid` that a user has participated in, to prevent duplicates.

paySet : A map storing for a given `pid` a tuple $(\{\gamma_i\}_{i\in[1,n]}, \mathcal{S}, \mathcal{R})$ where U is the set of containing channels and payment values, $\mathcal{S}$ is the set of all senders and $\mathcal{R}$ is the set of all receivers.

local : A map storing for a given `pid` a copy of all $\mathsf{tx}^{\mathsf{ep}}$ in a set $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$.

left : For each sender $\gamma_i.\mathsf{sender}$, a map storing for a given `pid` a tuple $(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ which contains the channel $\overline{\gamma}_i$ and corresponding state and refund transactions.

right : For each receiver $\gamma_i.\mathsf{receiver}$, a map storing for a given `pid` a tuple $(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}), \theta_{i,j})\}_{j\in[1,n]})$ which contains a channel and corresponding state transaction and the set of payment transactions. Along with each $\mathsf{tx}_{i,j}^{\mathsf{p}}$, a signature from the sender of the channel and the input of this transaction that comes from $\mathsf{tx}_j^{\mathsf{ep}}$ are saved.

sigSet : For each receiver $\gamma_i.\mathsf{receiver}$, a map, storing for a given `pid` the signatures for $\mathsf{tx}_i^{\mathsf{ep}}$ of all senders $\{\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}})\}_{j\in[1,n]}$.

Initialization

- Let τ be the current round.

dealer upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{CHANNELS\text{-}SET}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow} \mathcal{E}$

1. For all parties P_i in $\{\gamma_i.\mathsf{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookrightarrow} P_i$.

Each $\gamma_i.\mathsf{sender}$ and $\gamma_i.\mathsf{receiver}$

upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{init}, \{\gamma_j\}_{j\in[1,n]}) \overset{\tau+1}{\hookleftarrow}$ dealer

1. If `pid` $\in$ pidSet, abort. Add `pid` to pidSet, and let $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{P}$ be initially empty sets.
2. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{INIT\text{-}CHECK}, \{\gamma_j\}_{j\in[1,n]}) \overset{\tau+1}{\longrightarrow} \mathcal{E}$.

3. If $(\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]}) \xleftarrow{\tau+1} \mathcal{E}$, for each channel γ_j add $\gamma_j.\mathsf{sender}$ to $\mathcal{S}$ and $\gamma_j.\mathsf{receiver}$ to $\mathcal{R}$. Then set $\mathsf{paySet}(\texttt{pid}) := (\{\gamma_j\}_{j\in[1,n]}, \mathcal{S}, \mathcal{R})$ and $\mathcal{P} := \mathcal{R} \cup \mathcal{S}$. Otherwise abort.
4. Send $(\texttt{sid}, \texttt{pid}, \texttt{init-ok}) \xrightarrow{\tau+1} P_i$ to all $P_i \in \mathcal{P}$.
5. If $(\texttt{sid}, \texttt{pid}, \texttt{init-ok}) \xleftarrow{\tau+2} P_i$ from all parties in $\mathcal{P}$, go to the *Pre-Setup* phase. Otherwise abort.

Pre-Setup

- Let τ be the current round.

$\gamma_i.\mathsf{receiver}$

1. Let $\mathsf{tx}_i^{\mathsf{in}} := \texttt{GenTxIn}(\gamma_i.\mathsf{receiver}, \{\gamma_k\}_{k\in[1,n]})$.
2. Let $\mathsf{tx}_i^{\mathsf{ep}} := \texttt{GenTxEp}(\{\gamma_k\}_{k\in[1,n]}, \mathsf{tx}_i^{\mathsf{in}})$.
3. Send $(\texttt{sid}, \texttt{pid}, \texttt{pre-setup}, \mathsf{tx}_i^{\mathsf{ep}}) \xhookrightarrow{\tau} P_i$ for all $P_i \in \mathcal{P}$.

All users upon

$(\texttt{sid}, \texttt{pid}, \texttt{pre-setup}, \mathsf{tx}_i^{\mathsf{ep}}) \xleftarrow{\tau+1} \gamma_i.\mathsf{receiver}$ for all $i \in [1, n]$

1. For all $j \in [1,\ n]$, if $\texttt{CheckTxEp}(\mathsf{tx}_j^{\mathsf{ep}}, \gamma_j.\mathsf{receiver}, \{\gamma_k\}_{k\in[1,n]}) = \bot$, abort. otherwise set $\mathsf{local}(\texttt{pid}) = \{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}$ and go to the *Setup* phase.

Setup

- Let τ be the current round.

$\gamma_i.\mathsf{sender}$

1. Send $(\texttt{sid}, \texttt{pid}, \texttt{REQ-VALUE}, \gamma_i) \xhookrightarrow{\tau} \mathcal{E}$. If this message is replied by $(\texttt{sid}, \texttt{pid}, \texttt{VALUE}, \overline{\gamma}_i, \alpha_i) \xhookleftarrow{\tau} \mathcal{E}$, continue. Otherwise go idle.
2. Let $\mathsf{tx}_i^{\mathsf{state}} := \texttt{GenState}(\alpha_i, T, \overline{\gamma}_i)$.
3. Let $\mathsf{tx}_i^{\mathsf{r}} := \texttt{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{sender})$.

4. For all $j \in [1,n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to $\gamma_i.\mathsf{receiver}$, then create $\mathsf{tx}_{i,j}^{\mathsf{p}} := \mathtt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$.
5. Set $\mathsf{left}(\mathtt{pid}) := (\overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]})$.
6. Generate the set $\{\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})\}_{j\in[1,n]}$.
7. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}) \hookrightarrow^{\tau} \gamma_i.\mathsf{receiver}$.

$\gamma_i.\mathsf{receiver}$ upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}$

$, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}) \overset{\tau+1}{\longleftrightarrow} \gamma_i.\mathsf{sender}$

1. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{REQ\text{-}VALUE}, \gamma_i) \overset{\tau+1}{\longrightarrow} \mathcal{E}$. If this message is replied by $(\mathtt{sid}, \mathtt{pid}, \mathtt{VALUE}, \overline{\gamma}_i, \alpha_i) \overset{\tau+1}{\longleftarrow} \mathcal{E}$, continue. Otherwise go idle.
2. If $\mathsf{tx}_i^{\mathsf{state}} \neq \mathtt{GenState}(\alpha_i, T, \overline{\gamma}_i)$, abort.
3. For each element in $\{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}$, If $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})$ is not a correct signature, abort.
4. For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to $\gamma_i.\mathsf{receiver}$. If $\mathsf{tx}_{i,j}^{\mathsf{p}} \neq \mathtt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$, abort.
5. Set $\mathsf{right}(\mathtt{pid}) = (\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\overline{\gamma}_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}, \theta_{i,j})\}_{j\in[1,n]})$
6. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}ok}) \overset{\tau+1}{\longrightarrow} P_i$ for all $P_i \in \mathcal{P}$.

All users

1. If $(\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}ok}) \overset{\tau+2}{\longleftarrow} P_i$ for all $P_i \in \mathcal{R}$, go to the *Confirmation* phase. Otherwise abort.

Confirmation

- Let τ be the current round.

$\gamma_i.\mathsf{sender}$

1. Send $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \hookrightarrow^{\tau} \mathcal{F}_{Channel}$.

2. If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \xleftarrow{\tau+t_u} \mathcal{F}_{Channel}$, for all $j \in [1,n]$, create signature $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_j^{\mathsf{ep}})$ and send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_j^{\mathsf{ep}})) \xrightarrow{\tau+t_u} \gamma_j.\mathsf{receiver}$.

$\gamma_i.\mathsf{receiver}$ upon $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_j.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}}))$

$\xleftarrow{\tau+t_u+1} \gamma_j.\mathsf{sender}$ for all $j \in [1,n]$

1. If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}) \xleftarrow{\tau+t_u} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}, \overline{\gamma}_i) \xrightarrow{\tau+t_u+1} \mathcal{E}$. Otherwise abort.
2. If for all $j \in [1,n]$, $\sigma_{\gamma_j.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}})$ are valid signatures, let $\mathsf{sigSet} := \{(\sigma_{\gamma_j.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}}))\}_{j\in[1,n]}$. Otherwise abort.
3. Send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \xrightarrow{\tau+t_u+1} P_i$ for all $P_i \in \mathcal{P}$.

All users

1. If $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \xleftarrow{\tau+t_u+2} P_i$ for all $P_i \in \mathcal{R}$, go to the *Finalizing* phase. Otherwise abort.

Finalizing

- Let τ be the starting round.

$\gamma_i.\mathsf{sender}$

1. Let $\mathsf{tx}_i^{\mathsf{trans}} := \texttt{GenTrans}(\alpha_i, \overline{\gamma}_i)$.
2. Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output}) \hookrightarrow^{\tau} \mathcal{F}_{Channel}$.

$\gamma_i.\mathsf{receiver}$

1. If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output}) \xleftarrow{\tau+t_u} \mathcal{F}_{Channel}$, sign $\mathsf{tx}_i^{\mathsf{ep}}$ and add the signature to sigSet. $(\texttt{ssid}_L, \texttt{POST}, (\mathsf{tx}_i^{\mathsf{ep}}, \mathsf{sigSet})) \xrightarrow{\tau+t_u} \mathcal{G}_{Ledger}$.
2. Send $(\texttt{sid}, \texttt{pid}, \texttt{FINALIZED}, \overline{\gamma}_i) \xrightarrow{\tau+t_u} \mathcal{E}$.

Respond

Let t be the current round. Do the following:

γ_i.receiver at the end of every round t

1. For every pid in right.keyList(),
   let $(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}, \theta_{i,j})\}_{j\in[1,n]}) := \mathsf{right}(\mathtt{pid})$
   and let $\{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]} := \mathsf{local}(\mathtt{pid})$.
2. If $t < T - t_{\mathsf{c}} - 2\Delta$, one $\mathsf{tx}_j^{\mathsf{ep}}$ is on the ledger $\mathcal{L}$, and $\overline{\gamma}_i.\mathsf{st} = \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}$, do the following:
   a) Send $(\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$.
   b) If a transaction tx with $\mathsf{tx}.\mathsf{output} = \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output}$ is on $\mathcal{L}$ in round $t + t_{\mathsf{c}}$, wait Δ rounds.
   c) Sign $\mathsf{tx}_{i,j}^{\mathsf{p}}$ and set
   $\overline{\mathsf{tx}_{i,j}^{\mathsf{p}}} := (\mathsf{tx}_{i,j}^{\mathsf{p}}, \{\sigma_{\gamma_i.\mathsf{receiver}}(\mathsf{tx}_{i,j}^{\mathsf{p}}), \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})\})$.
   d) Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_{i,j}^{\mathsf{p}}}) \overset{t+t_{\mathsf{c}}+\Delta}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   e) When $\mathsf{tx}_{i,j}^{\mathsf{p}}$ appears on $\mathcal{L}$ in round $t_1 < T$, send
   $(\mathtt{sid}, \mathtt{pid}, \mathtt{PAID}, \overline{\gamma}_i) \overset{t_1}{\hookrightarrow} \mathcal{E}$

γ_i.sender at the end of every round t

1. For every pid in left.keyList(), let $(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}) := \mathsf{left}(\mathtt{pid})$.
2. If $t > T$ and a transaction tx with $\mathsf{tx}.\mathsf{output} = \mathsf{tx}_i^{\mathsf{state}}$ is on the ledger $\mathcal{L}$, but not any transaction in $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$, do the following:
   a) Sign $\mathsf{tx}_i^{\mathsf{r}}$ and set $\overline{\mathsf{tx}_i^{\mathsf{r}}} := (\mathsf{tx}_i^{\mathsf{r}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{r}}))$.
   b) Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}}) \overset{t}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   c) When $\mathsf{tx}_i^{\mathsf{r}}$ appears on $\mathcal{L}$ in round $t_1 < t + \Delta$, send
   $(\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}REFUND}, \overline{\gamma}_i) \overset{t_1}{\hookrightarrow} \mathcal{E}$

D.2.6 Proof

In this section, we present the simulator and formal proof that our multi-channel updates protocol Appendix D.2.5 UC-realizes the ideal functionality $\mathcal{F}_{update}$ Appendix D.2.4.

Simulator
Local variables:

enableSig : A map, sorting for a given $(\texttt{pid}, \mathsf{tx}_i^{\mathsf{ep}})$ the set of signatures $\{\sigma_{\gamma_j.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}})\}$ from all senders.

paySig : A map, sorting for a given $(\texttt{pid}, \mathsf{tx}_{i,j}^{\mathsf{p}})$ the signature $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})$.

Start phase

- Upon $(\texttt{sid}, \texttt{start}) \overset{\tau_s}{\hookleftarrow} \mathcal{F}_{update}$, Send $(\texttt{sid}, \texttt{start-ok}, t_{\mathsf{u}}, t_{\mathsf{c}}) \overset{\tau_s}{\hookrightarrow} \mathcal{F}_{update}$ and go to the *Initialization* phase.

Initialization phase

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{send-init}, \{\gamma_j\}_{j\in[1,n]}, \mathsf{dealer}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, for all corrupted $P_i \in \{\gamma_i.\mathsf{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$, send $(\texttt{sid}, \texttt{pid}, \texttt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookrightarrow} P_i$ on behalf of dealer.
- If the trigger party dealer is corrupted, upon $(\texttt{sid}, \texttt{pid}, \texttt{init}, \{\gamma_i\}_{i\in[1,n]}) \overset{\tau}{\hookleftarrow} \mathsf{dealer}$ on behalf on each honest party P_i, send $(\texttt{sid}, \texttt{pid}, \texttt{send-check}, \{\gamma_i\}_{i\in[1,n]}, P_i) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{send-init-ok}, \{\gamma_j\}_{j\in[1,n]}, P_i) \overset{\tau}{\hookrightarrow} \mathcal{S}$, for corrupted $P_j \in \{\gamma_i.\mathsf{sender}\}_{i\in[1,n]} \cup \{\gamma_i.\mathsf{receiver}\}_{i\in[1,n]}$, send $(\texttt{sid}, \texttt{pid}, \texttt{init-ok}) \overset{\tau}{\hookrightarrow} P_j$ on behalf of P_i.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{init-ok}) \overset{\tau+2}{\longleftrightarrow} P_j$ on behalf of P_i, where P_i is honest and P_j is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{init-acc}, P_i, P_j) \overset{\tau+2}{\longleftrightarrow} \mathcal{F}_{update}$.

Pre-Setup phase

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{presetup-req}, \gamma_i, \mathsf{tx}_x^{\mathsf{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where $\gamma_i.\mathsf{receiver}$ is a corrupted party, do the following.
  1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{pre-setup}, \mathsf{tx}_i^{\mathsf{ep}}) \overset{\tau+1}{\longleftrightarrow} \gamma_j.\mathsf{receiver}$ of behalf of P_i, where $\gamma_i.\mathsf{receiver}$ is corrupted, and P_i is honest, check if $\mathsf{tx}_i^{\mathsf{ep}} = \mathsf{tx}_x^{\mathsf{ep}}$, $(\texttt{sid}, \texttt{pid}, \texttt{presetup-acc}, P_i, \gamma_j.\mathsf{receiver}) \overset{\tau+1}{\longleftrightarrow} \mathcal{F}_{update}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{send-presetup}, \mathsf{tx}_i^{\mathsf{ep}}, \gamma_i.\mathsf{receiver}, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where $\gamma_i.\mathsf{receiver}$ is honest and P_j is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{pre-setup}, \mathsf{tx}_i^{\mathsf{ep}}) \overset{\tau}{\hookleftarrow} P_j$ on behalf of $\gamma_i.\mathsf{receiver}$.

Setup phase

- Upon, $(\texttt{sid}, \texttt{pid}, \texttt{send-setup-ok}, P_i, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where P_i is honest and P_j is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{setup-ok}) \overset{\tau}{\hookrightarrow} P_j$ on behalf of P_i.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}) \overset{\tau+1}{\longleftarrow} \gamma_i.\mathsf{sender}$, where $\gamma_i.\mathsf{sender}$ is corrupted, do the following.
  1. Check if any signature $\sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}})$ is not valid, abort.
  2. For all $j \in [1, n]$, let $\theta_{i,j}$ be the output of $\mathsf{tx}_j^{\mathsf{ep}}$ which corresponds to $\gamma_i.\mathsf{receiver}$. If $\mathsf{tx}_{i,j}^{\mathsf{p}} \neq \texttt{GenPay}(\mathsf{tx}_i^{\mathsf{state}}, \gamma_i.\mathsf{receiver}, \theta_{i,j})$, abort.
  3. Add the signature for each $\mathsf{tx}_{i,j}^{\mathsf{p}}$ to $\mathsf{paySig}(\texttt{pid}, \mathsf{tx}_{i,j}^{\mathsf{p}})$.
  4. $(\texttt{sid}, \texttt{pid}, \texttt{setup-acc}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}\}_{j\in[1,n]}) \overset{\tau+1}{\longrightarrow} \mathcal{F}_{update}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{send-setup}, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}, \gamma_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where $\gamma_i.\mathsf{sender}$ is honest but $\gamma_i.\mathsf{receiver}$ is corrupted, do the following.
  1. sign $\mathsf{tx}_{i,j}^{\mathsf{p}}$ on behalf of $\gamma_i.\mathsf{sender}$ and add it to $\mathsf{paySig}(\texttt{pid}, \mathsf{tx}_{i,j}^{\mathsf{p}})$.
  2. send $(\texttt{sid}, \texttt{pid}, \texttt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}) \overset{\tau}{\hookrightarrow} \gamma_i.\mathsf{receiver}$ on behalf of $\gamma_i.\mathsf{sender}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{setup-ok}) \overset{\tau+1}{\longleftarrow} \gamma_j.\mathsf{receiver}$ on behalf of P_i, where P_i is honest and $\gamma_j.\mathsf{receiver}$ is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{setup-finalized}, P_i, \gamma_j.\mathsf{receiver}) \overset{\tau+1}{\longrightarrow} \mathcal{F}_{update}$

Confirmation phase

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{send-sig}, \gamma_i.\mathsf{sender}, \gamma_j.\mathsf{receiver}, \mathsf{tx}_j^{\mathsf{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where $\gamma_i.\mathsf{sender}$ is honest but $\gamma_j.\mathsf{receiver}$ is corrupted, sign $\mathsf{tx}_j^{\mathsf{ep}}$ on behalf of $\gamma_i.\mathsf{sender}$ and send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_j^{\mathsf{ep}})) \overset{\tau}{\hookrightarrow} \gamma_j.\mathsf{receiver}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_j.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{ep}})) \overset{\tau}{\hookleftarrow} \gamma_j.\mathsf{sender}$ is received on behalf of $\gamma_i.\mathsf{receiver}$, where $\gamma_i.\mathsf{receiver}$ is honest and $\gamma_j.\mathsf{sender}$ is corrupted, check if all signatures are valid, send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-acc}, \gamma_i.\mathsf{receiver}, \gamma_j.\mathsf{sender}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.
- Upon, $(\texttt{sid}, \texttt{pid}, \texttt{send-confirmation-ok}, P_i, P_j) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where P_i is honest and P_j is corrupted, $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \overset{\tau+1}{\longrightarrow} P_j$ on behalf of P_i.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok}) \overset{\tau}{\hookleftarrow} \gamma_j.\mathsf{receiver}$ is received on behalf of an honest party P_i, where $\gamma_j.\mathsf{receiver}$ is corrupted, send $(\texttt{sid}, \texttt{pid}, \texttt{confirmation-finalized}, P_i, \gamma_i.\mathsf{receiver}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{update}$.

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{agg-sig}, \{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}, \mathcal{S}) \overset{\tau}{\hookleftarrow} \mathcal{S}$, for each $\mathsf{tx}_j^{\mathsf{ep}}$, sign the transaction on behalf of all honest $P_i \in \mathcal{S}$ and add $\sigma_{P_i}(\mathsf{tx}_j^{\mathsf{ep}})$ to $\mathsf{enableSig}(\texttt{pid}, \mathsf{tx}_j^{\mathsf{ep}})$

Finalizing phase

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-txep}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{ep}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where $\gamma_i.\mathsf{receiver}$ is a honest:
  1. Sign $\mathsf{tx}_i^{\mathsf{ep}}$ on behalf of $\gamma_i.\mathsf{receiver}$ and add the signature to $\mathsf{enableSig}(\texttt{pid}, \mathsf{tx}_i^{\mathsf{ep}})$
  2. Set $\overline{\mathsf{tx}_i^{\mathsf{ep}}} := (\mathsf{tx}_i^{\mathsf{ep}}, \mathsf{enableSig}(\texttt{pid}, \mathsf{tx}_i^{\mathsf{ep}}))$.
  3. Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{ep}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Respond phase

- Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma}_i, \mathsf{tx}_{i,j}^{\mathsf{p}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$, where $\gamma_i.\mathsf{receiver}$ is honest:
  1. Sign $\mathsf{tx}_{i,j}^{\mathsf{p}}$ on behalf of $\gamma_i.\mathsf{receiver}$ and add the signature to $\mathsf{paySig}(\texttt{pid}, \mathsf{tx}_{i,j}^{\mathsf{p}})$.
  2. Set $\overline{\mathsf{tx}_{i,j}^{\mathsf{p}}} := (\mathsf{tx}_{i,j}^{\mathsf{p}}, \mathsf{paySig}(\texttt{pid}, \mathsf{tx}_{i,j}^{\mathsf{p}})$.
  3. Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_{i,j}^{\mathsf{p}}}) \overset{\tau+t_c}{\longrightarrow} \mathcal{G}_{Ledger}$.
- Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{r}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{update}$ where $\gamma_i.\mathsf{sender}$ is honest:
  1. Sign $\mathsf{tx}_i^{\mathsf{r}}$ on behalf of $\gamma_i.\mathsf{sender}$ and set $\overline{\mathsf{tx}_i^{\mathsf{r}}} := (\mathsf{tx}_i^{\mathsf{r}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_i^{\mathsf{r}})$.
  2. Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}}) \overset{\tau+t_c}{\longrightarrow} \mathcal{G}_{Ledger}$.

Now, we show that in the view of the environment $\mathcal{E}$, a transcript resulting from interactions between the simulator $\mathcal{S}$ and the ideal functionality $\mathcal{F}_{update}$ is indistinguishable from a transcript resulting from an execution of the protocol Π in the presence of the adversary $\mathcal{A}$. Formally, we want to show that $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ and $\texttt{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}}$ are indistinguishable.

Our protocol Π and ideal functionality $\mathcal{F}_{update}$ both are executed in six phases: *Initialization*, *Pre-Setup*, *Setup*, *Confirmation*, *Finalize*, and *Respond*. For each phase separately, we show how the ideal world and the real world are indistinguishable for the environment.

In our description, we write $m[\tau]$ to denote that message m is observed at round τ. In other meaning, τ is the receiving round for message m (not the round it is sent). Moreover, sometimes we interact with ideal functionalities such as $\mathcal{F}_{Channel}$ and $\mathcal{G}_{Ledger}$. These functionalities in turn interact with either the environment $\mathcal{E}$ or other parties, who are possibly under adversarial, either by sending messages or additional impacts on publicly observable variables, i.e., the ledger $\mathcal{L}$. To capture this, we define $\mathsf{obsSet}(m, \mathcal{F}, \tau)$

as the set of all observable messages that are triggered by calling $\mathcal{F}$ with message m in round τ.

Lemma 20. *The initialization phase of protocol* Π *GUC-emulates the initialization phase of the functionality* $\mathcal{F}_{update}$.

Proof. Let τ be the starting round. Note that in the real world environment controls $\mathcal{A}$, and therefore, all corrupted parties. For better readability, we define the following messages that are used for *Initialization* phase in $\mathcal{F}_{update}$ and Π.

- $m_0 := (\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECK}, \{\gamma_i\}_{i\in[1,n]})$
- $m_1 := (\texttt{sid}, \texttt{pid}, \texttt{INIT-CHECKED}, \{\gamma_j\}_{j\in[1,n]})$
- $m_2 := (\texttt{sid}, \texttt{pid}, \texttt{CHANNELS-SET}, \{\gamma_i\}_{i\in[1,n]})$
- $m_3 := (\texttt{sid}, \texttt{pid}, \texttt{init}, \{\gamma_i\}_{i\in[1,n]})$
- $m_4 := (\texttt{sid}, \texttt{pid}, \texttt{init-ok})$
- $m_5 := (\texttt{sid}, \texttt{pid}, \texttt{send-init}, \{\gamma_j\}_{j\in[1,n]}, \mathsf{dealer})$
- $m_6 := (\texttt{sid}, \texttt{pid}, \texttt{send-check}, \{\gamma_i\}_{i\in[1,n]}, P_i)$
- $m_7 := (\texttt{sid}, \texttt{pid}, \texttt{send-init-ok}, \{\gamma_j\}_{j\in[1,n]}, P_i)$
- $m_8 := (\texttt{sid}, \texttt{pid}, \texttt{init-acc}, P_i, P_j)$

For each participant P_i, we compare messages that $\mathcal{E}$ receives from this party and the trigger party dealer in the ideal world and the real world. The types of messages depend on corruption cases for P_i and dealer. Note that messages from corrupted parties to $\mathcal{E}$ are not considered, because the environment is communicating with itself, which is trivially the same in the ideal and the real world.

Case 1: P_i **honest,** dealer **honest.**

Real world: $\mathcal{E}$ receives m_3 from dealer in round $\tau+1$ on behalf of all corrupted parties. Moreover, $\mathcal{E}$ receives m_0 from P_i, which contains the set of all channels in round $\tau+1$. If P_i gets m_1 from $\mathcal{E}$ in the response, then $\mathcal{E}$ receives m_4 from P_i on behalf of all corrupted parties in round $\tau+2$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau+1], m_0[\tau+1], m_4[\tau+2]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_5 to the simulator, which in turn, $\mathcal{S}$ sends m_3 on behalf on dealer to all corrupted parties in round τ. Moreover, $\mathcal{F}_{update}$ sends m_0 on behalf of P_i to $\mathcal{E}$ in round τ. Upon this message is replied by m_1 from $\mathcal{E}$, $\mathcal{F}_{update}$ sends m_7 to the simulator. After receiving this message, $\mathcal{S}$ sends m_4 to all corrupted parties on behalf of P_i in round $\tau+1$, which is received by $\mathcal{E}$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_3[\tau+1], m_0[\tau+1], m_4[\tau+2]\}$

Case 2: P_i honest, dealer corrupted.

Real world: Because dealer is corrupted, we do not need to consider messages from dealer to $\mathcal{E}$. Other received messages are similar to the previous case.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau+1], m_4[\tau+2]\}$

Ideal world: No longer $\mathcal{S}$ is required to send m_3 on behalf of dealer to $\mathcal{E}$. Simulation of the behavior of P_i is done the same as in the previous case.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_0[\tau+1], m_4[\tau+2]\}$

Case 3: P_i corrupted, dealer honest.

Real world: We do not to consider messages sent from P_i. $\mathcal{E}$ receives m_3 From dealer on behalf of all corrupted parties.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau+1]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_5 to the simulator, which in turn, $\mathcal{S}$ sends m_3 to all corrupted parties who are under the control of $\mathcal{E}$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_3[\tau+1]\}$

Lemma 21. *The pre-setup phase of protocol* Π *GUC-emulates the pre-setup phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Pre-Setup* phase in $\mathcal{F}_{update}$ and Π.

- $m_9 := (\texttt{sid}, \texttt{pid}, \texttt{pre-setup}, \mathsf{tx}_i^{\mathsf{ep}})$
- $m_{10} := (\texttt{sid}, \texttt{pid}, \texttt{presetup-req}, \gamma_i, \mathsf{tx}_i^{\mathsf{ep}})$
- $m_{11} := (\texttt{sid}, \texttt{pid}, \texttt{send-presetup}, \mathsf{tx}_i^{\mathsf{ep}}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{12} := (\texttt{sid}, \texttt{pid}, \texttt{presetup-acc}, P_i, P_j)$

In this phase, for each channel γ_i, $\mathcal{E}$ receives messages only from γ_i.receiver, so we should consider only one case. The case that γ_i.receiver is honest.

Real world: γ_i.receiver creates $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathsf{tx}_i^{\mathsf{ep}}$ and sends m_9 to all other parties, so this message is received by $\mathcal{E}$ on behalf of all corrupted parties in round $\tau+1$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_9[\tau+1]\}$

Ideal world: $\mathcal{F}_{update}$ first creates $\mathsf{tx}_i^{\mathsf{in}}$ and $\mathsf{tx}_i^{\mathsf{ep}}$ transactions for each channel γ_i. Then, $\mathcal{F}_{update}$ sends m_{11} to the simulator for all corrupted parties P_j. When $\mathcal{S}$ receives this

message, sends m_9 to all corrupted parties on behalf of $\gamma_i.\mathsf{receiver}$. The messages are received by $\mathcal{E}$ in round $\tau + 1$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_9[\tau + 1]\}$

Lemma 22. *The setup phase of protocol* Π *GUC-emulates the setup phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Setup* phase in $\mathcal{F}_{update}$ and Π.

- $m_{13} := (\mathtt{sid}, \mathtt{pid}, \mathtt{REQ\text{-}VALUE}, \gamma_i)$
- $m_{14} := (\mathtt{sid}, \mathtt{pid}, \mathtt{VALUE}, \overline{\gamma}_i, \alpha_i)$
- $m_{15} := (\mathtt{sid}, \mathtt{pid}, \mathtt{setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$
- $m_{16} := (\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}ok})$
- $m_{17} := (\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}setup}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]})$
- $m_{18} := (\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}acc}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{state}}, \{\mathsf{tx}_{i,j}^{\mathsf{p}}\}\}_{j\in[1,n]})$
- $m_{19} := (\mathtt{sid}, \mathtt{pid}, \mathtt{send\text{-}setup\text{-}ok}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{20} := (\mathtt{sid}, \mathtt{pid}, \mathtt{setup\text{-}finalized}, P_i, P_j)$

In this phase, for each channel γ_i, both the sender and the receiver have interactions with the environment. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: $\gamma_i.\mathsf{sender}$ **honest,** $\gamma_i.\mathsf{receiver}$ **honest.**

Real world: $\gamma_i.\mathsf{sender}$ sends m_{13} to $\mathcal{E}$ in round τ. Upon this message is replied by m_{14}, $\gamma_i.\mathsf{sender}$ generates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and the set $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. Then she sends m_{15} to $\gamma_i.\mathsf{receiver}$. When $\gamma_i.\mathsf{receiver}$ gets this message, first asks $\mathcal{E}$ about the payment value via message m_{13} in round $\tau + 1$. Upon this message is replied by m_{14}, $\gamma_i.\mathsf{receiver}$ checks validity of the transactions inside received m_{15}, and then sends m_{16} to all other parties, which is received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau + 2$. Note that two m_{13} messages are received by $\mathcal{E}$ in different rounds. One from the sender and one from the receiver.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{13}[\tau + 1], m_{16}[\tau + 2]\}$

Ideal world: $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ on behalf of $\gamma_i.\mathsf{sender}$ in round τ. After receiving the response m_{14}, $\mathcal{F}_{update}$ creates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and the set $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. Again, $\mathcal{F}_{update}$

sends m_{13} to $\mathcal{E}$ this time on behalf of γ_i.receiver in round $\tau+1$. After receiving the response, $\mathcal{F}_{update}$ sends m_{19} to the simulator, which in turn, $\mathcal{S}$ sends m_{16} to all corrupted parties, which is received in round $\tau+2$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau], m_{13}[\tau+1], m_{16}[\tau+2]\}$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted.**

Real world: In this case, we only consider messages that are sent from the sender. Similar to the previous case, γ_i.sender sends m_{13} to $\mathcal{E}$ in round τ, and waits for the response m_{14}. Then she generates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and the set $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$ and sends m_{15} to γ_i.receiver. This time the message m_{15} is observed by $\mathcal{E}$ in round $\tau+1$. because the receiver is corrupted.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{15}[\tau+1]\}$

Ideal world: Similar to the previous case, $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ on behalf of γ_i.sender in round τ. After receiving the response m_{14}, $\mathcal{F}_{update}$ creates $\mathsf{tx}_i^{\mathsf{state}}$, $\mathsf{tx}_i^{\mathsf{r}}$, and the set $\{\mathsf{tx}_{i,j}^{\mathsf{p}}\}_{j\in[1,n]}$. This time $\mathcal{F}_{update}$ sends m_{17} to the simulator, which in turn, $\mathcal{S}$ sends m_{15} to the corrupted receiver in round τ.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau], m_{15}[\tau+1]\}$

Case 3: γ_i.sender **corrupted,** γ_i.receiver **honest.**

Real world: In this case, we only consider messages that are sent from the receiver. At first, When γ_i.receiver gets m_{15} message from the sender, sends m_{13} to $\mathcal{E}$ to get the payment value in round $\tau+1$. Then, this party after receiving the response from $\mathcal{E}$, checks the validity of the transactions inside m_{15}. Finally, she sends m_{16} to all other parties, which is received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau+2$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau+1], m_{16}[\tau+2]\}$

Ideal world: $\mathcal{S}$ gets transactions $\mathsf{tx}_i^{\mathsf{state}}$, and the set $\{(\mathsf{tx}_{i,j}^{\mathsf{p}}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_{i,j}^{\mathsf{p}}))\}_{j\in[1,n]}$ from $\mathcal{A}$ and sends them to $\mathcal{F}_{update}$ via m_{18} if they are correct. $\mathcal{F}_{update}$ sends m_{13} to $\mathcal{E}$ this time on behalf of γ_i.receiver in round $\tau+1$. If this message is reponsed by $\mathcal{E}$ with m_{14}, $\mathcal{F}_{update}$ checks correctness of $\mathsf{tx}_i^{\mathsf{state}}$ received from the simulator. $\mathcal{F}_{update}$ sends m_{19} to the simulator, which in turn, $\mathcal{S}$ sends m_{16} to all corrupted parties in round $\tau+1$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau+1], m_{16}[\tau+2]\}$

Lemma 23. *The confirmation phase of protocol* Π *GUC-emulates the confirmation phase of the functionality* $\mathcal{F}_{update}$*.*

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{21} := (\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output})$
- $m_{22} := (\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{state}}.\mathsf{output})$
- $m_{23} := (\texttt{sid}, \texttt{pid}, \texttt{confirmation}, \sigma_{\gamma_i.\mathsf{sender}}(\mathsf{tx}_j^{\mathsf{ep}}))$
- $m_{24} := (\texttt{sid}, \texttt{pid}, \texttt{OPENED}, \overline{\gamma}_i)$
- $m_{25} := (\texttt{sid}, \texttt{pid}, \texttt{confirmation-ok})$
- $m_{26} := (\texttt{sid}, \texttt{pid}, \texttt{send-sig}, \gamma_i.\mathsf{sender}, \gamma_j.\mathsf{receiver}, \mathsf{tx}_j^{\mathsf{ep}})$
- $m_{27} := (\texttt{sid}, \texttt{pid}, \texttt{confirmation-acc}, \gamma_i.\mathsf{receiver}, \gamma_j.\mathsf{sender})$
- $m_{28} := (\texttt{sid}, \texttt{pid}, \texttt{send-confirmation-ok}, \gamma_i.\mathsf{receiver}, P_j)$
- $m_{29} := (\texttt{sid}, \texttt{pid}, \texttt{confirmation-finalized}, P_i, P_j)$
- $m_{30} := (\texttt{sid}, \texttt{pid}, \texttt{agg-sig}, \{\mathsf{tx}_j^{\mathsf{ep}}\}_{j\in[1,n]}, \mathcal{S})$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: γ_i.sender **honest,** γ_i.receiver **honest.**

Real world: γ_i.sender sends m_{21} to $\mathcal{F}_{Channel}$ in round τ to update the state of $\overline{\gamma}_i$ using $\mathsf{tx}_i^{\mathsf{state}}$. If the update is executed correctly, γ_i.sender sends m_{23} to each receiver. This message is received by $\mathcal{E}$ on behalf of each corrupted receiver in round $\tau + t_u + 1$. Again, if the update is executed correctly, γ_i.receiver waits until receiving signatures to $\mathsf{tx}_i^{\mathsf{ep}}$ from all senders. Then, she sends m_{24} to $\mathcal{E}$ in round $\tau + t_u + 1$. Also, after verifying all signatures, she sends m_{25} messages to all parties, which are received by $\mathcal{E}$ on behalf of corrupted parties in round $\tau + t_u + 2$.

$$\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{23}[\tau+t_u+1], m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{F}_{Channel}, \tau)\}$$

Ideal world: $\mathcal{F}_{update}$ sends m_{21} massage to $\mathcal{F}_{Channel}$. If the update is executed correctly, $\mathcal{F}_{update}$ via message m_{26}, asks $\mathcal{S}$ to generate a signature to each $\mathsf{tx}_j^{\mathsf{ep}}$ on behalf of γ_i.sender and sends it to the corresponding receiver if the receiver is corrupted. This is done via message m_{23} which is received by $\mathcal{E}$ in round $\tau + t_u + 1$. Moreover, $\mathcal{F}_{update}$ sends m_{24} to $\mathcal{E}$ in round $\tau + t_u + 1$ and m_{28} to the simulator, which in turn, $\mathcal{S}$ sends m_{25} on behalf of γ_i.receiver to all corrupted parties, which is received by $\mathcal{E}$ in round $\tau + t_u + 2$.

$$\texttt{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{23}[\tau+t_u+1], m_{24}[\tau+t_u+1], m_{25}[\tau+t_u+2]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{F}_{Channel}, \tau)\}$$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted.**

Real world: In this case, we only consider messages that are sent from the sender. γ_i.sender sends m_{21} to $\mathcal{F}_{Channel}$ in round τ. If the update is executed correctly, she sends

m_{23} to each receiver. This message is received by $\mathcal{E}$ in behalf of each corrupted receiver in round $\tau + t_u + 1$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{23}[\tau + t_u + 1]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{F}_{Channel}, \tau)$

Ideal world: Again, $\mathcal{F}_{update}$ sends m_{21} massage to $\mathcal{F}_{Channel}$ and if the update is executed correctly, $\mathcal{F}_{update}$ sends m_{26} to $\mathcal{S}$ to generate a signature to each $\mathsf{tx}_j^{\mathsf{ep}}$ on behalf of $\gamma_i.\mathsf{sender}$. Then $\mathcal{S}$ sends it to the corresponding receiver if she is corrupted via message m_{23} in round $\tau + t_u$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{23}[\tau + t_u + 1]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{F}_{Channel}, \tau)$

Case 3: $\gamma_i.\mathsf{sender}$ **corrupted,** $\gamma_i.\mathsf{receiver}$ **honest.**

Real world: In this case, we only consider messages that are sent from the receiver. If the update is executed correctly, $\gamma_i.\mathsf{receiver}$ verifies received signatures to $\mathsf{tx}_i^{\mathsf{ep}}$ from all senders, sends m_{24} to $\mathcal{E}$ in round $\tau + t_u + 1$, and sends m_{25} messages to all parties in round $\tau + t_u + 2$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{24}[\tau + t_u + 1], m_{25}[\tau + t_u + 2]\}$

Ideal world: $\mathcal{S}$ receives signatures form a corrupted sender. If the signature is valid $\mathcal{S}$ sends m_{27} to $\mathcal{F}_{update}$. If the update has already executed correctly, then $\mathcal{F}_{update}$ sends m_{24} to $\mathcal{E}$ in round $\tau + t_u + 1$. Moreover, sends m_{28} to the simulator, which in turn, $\mathcal{S}$ sends m_{25} on behalf of $\gamma_i.\mathsf{receiver}$ to all corrupted parties in round $\tau + t_u + 1$.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{24}[\tau + t_u + 1], m_{25}[\tau + t_u + 2]\}$

Lemma 24. *The finalizing phase of protocol* Π *GUC-emulates the finalizing phase of the functionality* $\mathcal{F}_{update}$*.*

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{31} := (\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output})$
- $m_{32} := (\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma}_i.\mathsf{id}, \mathsf{tx}_i^{\mathsf{trans}}.\mathsf{output})$
- $m_{33} := (\texttt{ssid}_L, \texttt{POST}, (\mathsf{tx}_i^{\mathsf{ep}}, \mathsf{sigSet}))$
- $m_{34} := (\texttt{sid}, \texttt{pid}, \texttt{FINALIZED}, \overline{\gamma}_i)$
- $m_{35} := (\texttt{sid}, \texttt{pid}, \texttt{post-texp}, \overline{\gamma}_i, \mathsf{tx}_i^{\mathsf{ep}})$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$. We need to consider different corruption cases for these parties except the case that both of them are corrupted.

Case 1: γ_i.sender **honest,** γ_i.receiver **honest.**

Real world: γ_i.sender generates $\mathsf{tx}_i^{\mathsf{in}}$, which transfers α_i coins from the sender to the receiver. Then, sends m_{31} to $\mathcal{F}_{Channel}$ in round τ. If the update fails, the receiver sends m_{33} to $\mathcal{G}_{Ledger}$ in round $\tau + t_u$ and post $\mathsf{tx}_i^{\mathsf{ep}}$ to the ledger. Finally, γ_i.receiver sends m_{34} to $\mathcal{E}$ in round $\tau + t_u$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{31}, \mathcal{F}_{Channel}, \tau) \cup$
$\mathsf{obsSet}(m_{33}, \mathcal{G}_{Ledger}, \tau + t_u)$

Ideal world: $\mathcal{F}_{update}$ generates $\mathsf{tx}_i^{\mathsf{in}}$ and updates the channel γ_i via sending m_{31} to $\mathcal{F}_{Channel}$ in round τ. After the update execution, $\mathcal{F}_{update}$ sends m_{34} to $\mathcal{E}$ in round $\tau + t_u$ and on behalf of the receiver. If the update fails, $\mathcal{F}_{update}$ sends m_{35} to $\mathcal{S}$ and asks it to post $\mathsf{tx}_i^{\mathsf{ep}}$ on the ledger via message m_{33} to $\mathcal{G}_{Ledger}$ in round $\tau + t_u$ on behalf of γ_i.receiver.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{31}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{Ledger}, \tau + t_u)$

Case 2: γ_i.sender **honest,** γ_i.receiver **corrupted.**

Real world: In this case, we ignore messages that are sent directly from the receiver to $\mathcal{E}$. γ_i.sender generates $\mathsf{tx}_i^{\mathsf{in}}$, and sends m_{33} to $\mathcal{F}_{Channel}$ to update the channel.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{33}, \mathcal{F}_{Channel}, \tau)$

Ideal world: $\mathcal{F}_{update}$ generates $\mathsf{tx}_i^{\mathsf{in}}$ and updates the channel γ_i via sending m_{33} to $\mathcal{F}_{Channel}$ in round τ.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{33}, \mathcal{F}_{Channel}, \tau)$

Case 3: γ_i.sender **corrupted,** γ_i.receiver **honest.**

Real world: In this case, we only consider messages that are sent from the receiver. γ_i.receiver waits until time $\tau + t_u$. If message m_{32} is received in this round, the final transfer has been performed, so γ_i.receiver sends m_{34} to $\mathcal{E}$. If m_{32} is not received and the update fails, sends m_{33} to $\mathcal{G}_{Ledger}$ in round $\tau + t_u$.

$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{Ledger}, \tau + t_u)$

Ideal world: $\mathcal{F}_{update}$ waits until receiving m_{32} from $\mathcal{F}_{Channel}$. If this happens, the update is executed and $\mathcal{F}_{update}$ sends m_{34} to $\mathcal{E}$ on behalf of the receiver in round $\tau + t_u$. Otherwise, $\mathcal{F}_{update}$ sends m_{35} to $\mathcal{S}$ and asks it to send m_{33} to $\mathcal{G}_{Ledger}$ on behalf of the receiver.

$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{34}[\tau + t_u]\} \cup \mathsf{obsSet}(m_{33}, \mathcal{G}_{Ledger}, \tau + t_u)$

Lemma 25. *The respond phase of protocol* Π *GUC-emulates the respond phase of the functionality* $\mathcal{F}_{update}$.

Proof. Again we compare observed messages by $\mathcal{E}$ in the ideal world and the real world. Let τ be the starting round, and consider the following definitions for all messages that are used for *Confirmation* phase in $\mathcal{F}_{update}$ and Π.

- $m_{36} := (\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id})$
- $m_{37} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i,j}})$
- $m_{38} := (\texttt{sid}, \texttt{pid}, \texttt{PAID}, \overline{\gamma}_i)$
- $m_{39} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i})$
- $m_{40} := (\texttt{sid}, \texttt{pid}, \texttt{FORCE-REFUND}, \overline{\gamma}_i)$
- $m_{41} := (\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma}_i, \mathsf{tx}^{\mathsf{p}}_{i,j})$
- $m_{42} := (\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma}_i, \mathsf{tx}^{\mathsf{r}}_i)$

For each channel γ_i, both the sender and the receiver send messages to $\mathcal{E}$ independently. We consider cases where the parties are honest.

Case 1: γ_i.receiver **honest, Pay.**

Real world: In every round, γ_i.receiver checks whether one of transactions in $\{\mathsf{tx}^{\mathsf{ep}}_j\}_{j\in[1,n]}$ is observed on the ledger and $\tau < T - t_c - 2\Delta$. If so, she closes the channel γ_i via message m_{36} to $\mathcal{F}_{Channel}$. When the channel becomes closed and $\mathsf{tx}^{\mathsf{state}}_i$ is found on the ledger, γ_i.receiver waits time Δ, and then, post transaction $\mathsf{tx}^{\mathsf{p}}_{i,j}$, which forces the payment. This is done by sending m_{37} to $\mathcal{G}_{Ledger}$. The receiver finally sends m_{38} to $\mathcal{E}$ in round $\tau + t_c + 2\Delta$

$$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{38}[\tau + t_c + 2\Delta]\} \cup \mathsf{obsSet}(m_{36}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{37}, \mathcal{G}_{Ledger}, \tau + t_c + \Delta)$$

Ideal world: In every round, $\mathcal{F}_{update}$ checks if one of transactions in $\{\mathsf{tx}^{\mathsf{ep}}_j\}_{j\in[1,n]}$ is observed on the ledger and $\tau < T - t_c - 2\Delta$, sends m_{36} to $\mathcal{F}_{Channel}$ to close the channel γ_i. After a successful closure, $\mathcal{F}_{update}$ after a time Δ, send m_{41} to the simulator. The $\mathcal{S}$ aggregates signatures required for spending $\mathsf{tx}^{\mathsf{p}}_{i,j}$ and sends m_{37} to $\mathcal{G}_{Ledger}$. When this transaction appears on the ledger, $\mathcal{F}_{update}$ sends m_{38} to $\mathcal{E}$.

$$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{38}[\tau + t_c + 2\Delta]\} \cup \mathsf{obsSet}(m_{36}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{37}, \mathcal{G}_{Ledger}, \tau + t_c + \Delta)$$

Case 2: γ_i.sender **honest, Revoke.**

Real world: In every round, when τ is larger than T and channel γ_i has been closed, but not any payment transaction $\mathsf{tx}^{\mathsf{p}}_{i,j}$ is on the ledger, γ_i.sender signs $\mathsf{tx}^{\mathsf{r}}_i$ and post it on the ledger via message m_{39} to $\mathcal{G}_{Ledger}$. After observing $\mathsf{tx}^{\mathsf{r}}_i$ on the ledger, γ_i.sender sends m_{40} to $\mathcal{E}$.

$$\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{40}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{39}, \mathcal{G}_{Ledger}, \tau)$$

Ideal world: In every round, when τ is larger than T and channel γ_i has been closed, $\mathcal{F}_{update}$ sends m_{42} to the simulator, which in turn, $\mathcal{S}$ sign $\mathsf{tx}^{\mathsf{r}}_i$ on behalf of γ_i.sender and sends m_{39} to $\mathcal{G}_{Ledger}$. When $\mathsf{tx}^{\mathsf{r}}_i$ is observed on the ledger, $\mathcal{F}_{update}$ sends m_{40} to $\mathcal{E}$ again on behalf of γ_i.sender.

$$\text{EXEC}_{\mathcal{F}_{update},\mathcal{S},\mathcal{E}} := \{m_{40}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{39}, \mathcal{G}_{Ledger}, \tau)$$

Theorem 14. *For ideal functionalities $\mathcal{F}_{Channel}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, and $\mathcal{G}_{Ledger}$ and for any $T, \Delta \in \mathbb{N}$, the protocol Π GUC-emulates the the functionality $\mathcal{F}_{update}$.*

This theorem follows directly from Lemmas 20 to 25.

D.3 Discussion on security and privacy

In Section 5.3.1, we introduced the security and privacy goals of interest, atomicity, and strong value privacy. In Section 5.5.3, we informally showed that the security and privacy goals are achieved by our construction. Further, in Appendix D.2.4 we defined an ideal functionality $\mathcal{F}_{update}$ for multi-channel updates, and then we proved that the Thora protocol GUC-emulates the ideal functionality. In this section, formalize our security and privacy properties and then prove that $\mathcal{F}_{update}$ fulfills them.

D.3.1 Atomicity

For our multi-channel updates, let $U := \{(\gamma_i, \alpha_i)\}_{i\in[1,n]}$ be the set of updates. Each tuple (γ_i, α_i) contains a channel γ_i, which will be updated, and a value α_i which determines the update amount of that channel. For each channel γ_i, we define the following possible outcomes. We define γ_i as *successful* if α_i coins have been transferred from the sender to the receiver. I.e., $\gamma_i.\mathsf{balance}(\gamma_i.\mathsf{sender})$ has been decreased by α_i and additionally $\gamma_i.\mathsf{balance}(\gamma_i.\mathsf{receiver})$ has been increased by α_i at the end of the protocol execution. We define γ_i as *reverted* if, at the end of the protocol execution, the channel balance is the same as at the start of the protocol execution. A successful or reverted channel γ_i can be *compensated* if one of the users is malicious and deviates from protocol at the cost of losing her funds to the neighboring user without affecting the security of other users. We define γ_i as *punished* if there is an honest node that receives the total channel balance via the channel punishment mechanism. For every other outcome, we say a channel is *invalid*. A channel can have multiple outcomes, e.g., reverted and compensated.

Now, we define a security game $\mathtt{Atom}_{\mathcal{A},\Pi}$ as follows. The adversary $\mathcal{A}$ selects a set of n channels $\{\gamma_1, \gamma_2, ..., \gamma_n\}$, chooses the corrupted users from the users of these channels, selects dealer and sends these values to the challenger. The challenger sets `sid` and `pid` to two random identifiers. With these parameters, the challenger starts running Thora from the *Initialization* phase on the input of the channels set for the given dealer. The behavior of honest parties can be simulated directly by the challenger, and every time a corrupted party needs to be contacted, the challenger sends the query to $\mathcal{A}$ and waits for the corresponding answer. $\mathcal{A}$ can respond correctly, wrongly, not at all, manipulate the ledger by posting (valid) transactions, updating channels, etc.

After the protocol execution terminates, we say that $\mathcal{A}$ wins if one of the following cases holds after the execution.

1. There exist two channels γ_i, γ_j, each with at least one honest user, where γ_i is successful, and γ_j is reverted, and none of the channels are compensated.

2. There exists any channel γ_i without two corrupted nodes such that γ_i is invalid or channel γ_j with two honest users such that γ_j is punished.

Definition 15. We say that a multi-channel updates protocol achieves atomicity if, for every PPT adversary $\mathcal{A}$, the adversary wins the $\texttt{Atom}_{\mathcal{A},\Pi}$ game with negligible probability.

Theorem 15. *The multi-channel updates functionality $\mathcal{F}_{update}$ achieves atomicity property defined in Definition 15.*

Proof. Assume that there is an adversary $\mathcal{A}$ that can win the game $\texttt{Atom}_{\mathcal{A},\Pi}$, which implies that at least one of the two conditions (1) or (2) from the game definitions holds.

Suppose that (1) holds. We have two possible scenarios. First, $\mathcal{F}_{update}$ has created $\mathsf{tx}_i^{\mathsf{trans}}$ in the *Finalizing* phase, and has updated the channel γ_i using $\mathsf{tx}_i^{\mathsf{trans}}$ successfully. Second, at least one $\mathsf{tx}_k^{\mathsf{ep}}$ and $\mathsf{tx}_{i,k}^{\mathsf{p}}$ are on the ledger.

If we are in the first case, both γ_i and γ_j have been entered into the Finalizing phase of $\mathcal{F}_{update}$ because both have at least one honest user. If $\gamma_j.\mathsf{receiver}$ is honest, $\mathcal{F}_{update}$ forces the payment of γ_j either by updating with $\mathsf{tx}_j^{\mathsf{trans}}$ or posting $\mathsf{tx}_j^{\mathsf{ep}}$ in the finalizing phase. Note that if one $\mathsf{tx}^{\mathsf{ep}}$ appears on the ledge, as $\gamma_j.\mathsf{receiver}$ is honest, $\mathcal{F}_{update}$ forces the payment in the response phase.

Now consider the case that $\gamma_j.\mathsf{receiver}$ is malicious. By the assumption $\gamma_j.\mathsf{sender}$ is honest. As $\mathcal{F}_{update}$ has started the finalizing phase, $\mathsf{tx}_j^{\mathsf{trans}}$ should be generated, and γ_j should be tried to be updated using $\mathsf{tx}_j^{\mathsf{trans}}$ unless $\gamma_j.\mathsf{receiver}$ does not cooperate in the updating. In this case, γ_j will be compensated.

If we are in the second case, if $\gamma_j.\mathsf{receiver}$ is honest, $\mathcal{F}_{update}$ forces the payment on behalf of her in the response phase. If $\gamma_j.\mathsf{receiver}$ is malicious, she has refused to force the payment by posting $\mathsf{tx}_{k,j}^{\mathsf{p}}$ and γ_j would be compensated. It follows that (1) cannot hold.

Similarly, (2) cannot hold: The only possible outcomes that the ideal functionality allows for channels with at least one honest node are successful, reverted, or compensated. Since both (1) and (2) cannot hold, it follows that such an adversary does not exist. □

D.3.2 Strong value privacy

For a protocol Π and an adversary $\mathcal{A}$, we define another game $\texttt{VPriv}$ to capture the strong value privacy property. $\mathcal{A}$ selects dealer, and chooses a set of n channels $\{\gamma_1, \gamma_2, ..., \gamma_n\}$, where for each channel γ_i both $\gamma_i.\mathsf{receiver}$ and $\gamma_i.\mathsf{sender}$ are honest or semi-honest parties. In other words, corrupted parties involved in the protocol do not deviate from the protocol during the execution. The goal $\mathcal{A}$ is to guess the payment values regarding the channels with both honest senders and honest receivers. $\mathcal{A}$ has access to messages sent from honest parties to corrupted ones and publicly auditable parameters, like transactions posted to the ledger.

$\mathcal{A}$ sends the set of channels to the challenger. The challenger sets $\texttt{sid}$ and $\texttt{pid}$ to two random identifiers. Then, the challenger starts simulating Thora from the *Initialization*

phase on the input of the channels set for the given dealer. We assume that messages honest parties receive from $\mathcal{E}$ about the payment values (REQ-VALUE messages) are not leaked to any other parties. Moreover, we assume the values $\mathcal{E}$ sends to the receiver and the sender of a single channel are the same.

By the end of the protocol simulation, $\mathcal{A}$ sends the set $\{\alpha'_{i_1}, \alpha'_{i_2}, ..., \alpha'_{i_k}\}$ to the challenger, each α'_{i_j} is the guess of $\mathcal{A}$ for the payment value in channel γ_{i_j} where both the sender and the receiver are honest. We say that $\mathcal{A}$ wins the game if there is at least one $j \in [1, k]$ such that $\alpha'_{i_j} = \alpha_{i_j}$.

Definition 16. We say that a multi-channel updates protocol achieves strong value privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the $\mathtt{VPriv}_{\mathcal{A},\Pi}$ game with negligible probability.

Theorem 16. *The multi-channel updates functionality $\mathcal{F}_{update}$ achieves the strong value privacy property.*

Proof. We assume that k is negligible with regard to the size of the domain which payment values can be chosen from. Thus, without any leaked information about payment values, the probability of the adversary winning the game is negligible.

Suppose that there is an adversary $\mathcal{A}$ that can win the game $\mathtt{VPriv}_{\mathcal{A},\Pi}$ with a non-negligible probability. It means that there is a payment value α_{i_j}, where $\mathcal{A}$ is able to extract some information about the value and guess α'_{i_j}, such that $\alpha'_{i_j} = \alpha_{i_j}$. The only ways to get information about α_{i_j} are the messages $\mathcal{F}_{update}$ sends to corrupted parties and transactions that are posted to the ledger.

α_{i_j} is encoded only in four types of transactions. $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\{\mathsf{tx}^{\mathsf{p}}_{i_j,k}\}_{k\in[1,n]}$, $\mathsf{tx}^{\mathsf{r}}_{i_j}$, and $\mathsf{tx}^{\mathsf{trans}}_{i_j}$. $\gamma_{i_j}.\mathsf{sender}$ is honest so all these transactions are created by $\mathcal{F}_{update}$. $\mathsf{tx}^{\mathsf{r}}_{i_j}$ and $\mathsf{tx}^{\mathsf{trans}}_{i_j}$ are never sent to other parties inside exchanged messages. Moreover, because $\gamma_{i_j}.\mathsf{receiver}$ is honest, $\mathcal{F}_{update}$ will not sent $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\mathsf{tx}^{\mathsf{p}}_{i_j,k}$ neither to $\gamma_{i_j}.\mathsf{receiver}$ nor other parties.

On the other hand, since all parties are honest or semi-honest and do not deviate from the protocol, we expect the final update using transaction $\mathsf{tx}^{\mathsf{trans}}_{i_j}$ to be executed successfully for all channels, and no $\mathsf{tx}^{\mathsf{ep}}$ is required to be posted on the ledger. Therefore, in the *respond* phase, $\mathsf{tx}^{\mathsf{state}}_{i_j}$, $\mathsf{tx}^{\mathsf{p}}_{i_j,k}$, or $\mathsf{tx}^{\mathsf{r}}_{i_j}$ are not required to be posted on the ledger, and $\mathcal{A}$ has no way to observe these transactions. □

APPENDIX E

Appendix to Chapter 6

E.1 On the usage of the UC-Framework

To formally model the security of our construction, we use a synchronous version of the global UC framework (GUC) [CDPW07] which extends the standard UC framework [Can01] by allowing for a global setup. Since our model is essentially the same as in [AEE+21], which in turn follows [DFH18, DEF+19b], parts of this section are taken verbatim from there.

Protocols and adversarial model. We consider a protocol π that runs between parties from the set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ that takes as input a security parameter 1^λ (with $\lambda \in \mathbb{N}$) and an auxiliary input $z \in \{0,1\}^*$, and who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). By corruption, we mean that $\mathcal{A}$ takes full control over P_i and learns its internal state. Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{E}$ – which represents anything "external" to the current protocol execution. The environment also observes all outputs returned by the parties of the protocol. In addition to the above entities, the parties can have access to ideal functionalities $\mathcal{H}_1, \ldots, \mathcal{H}_m$. In this case we say that the protocol π *works in the* $(\mathcal{H}_1, \ldots, \mathcal{H}_m)$*-hybrid model* and write $\pi^{\mathcal{H}_1,\ldots,\mathcal{H}_m}$.

Modeling time and communication. We assume a synchronous communication network, which means that the execution of the protocol happens in rounds. Let us emphasize that the notion of rounds is just an abstraction which simplifies our model and allows us to argue about the time complexity of our protocols in a natural way. We follow [DEF+19b], which in turn follows [KMTZ13], and formalize the notion of rounds via an ideal functionality $\widehat{\mathcal{F}}_{clock}$ representing "the clock". On a high level, the ideal functionality requires all honest parties to indicate that they are prepared to proceed to the next round before the clock is "ticked". We treat the clock functionality as a *global*

ideal functionality using the GUC model. This means that all entities are always aware of the given round.

We assume that parties of a protocol are connected via authenticated communication channels with guaranteed delivery of exactly one round. This means that if a party P sends a message m to party Q in round t, party Q receives this message in the beginning of round $t+1$. In addition, Q is sure that the message was sent by party P. The adversary can see the content of the message and can reorder messages that were sent in the same round. However, it can not modify, delay or drop messages sent between parties, or insert new messages. The assumptions on the communication channels are formalized as an ideal functionality $\mathcal{F}_{GDC}$. We refer the reader to [DEF+19b] its formal description.

While the communication between two parties of a protocol takes exactly one round, all other communication – for example, between the adversary $\mathcal{A}$ and the environment $\mathcal{E}$ – takes zero rounds. For simplicity, we assume that any computation made by any entity takes zero rounds as well.

Handling coins. We model the money mechanics offered by UTXO cryptocurrencies, such as Bitcoin, via a *global* ideal functionality $\mathcal{L}$ using the GUC model. Our functionality is parameterized by a *delay parameter* Δ which upper bounded in the maximal number of rounds it takes to publish a valid transaction, and a signature scheme Σ. The functionality accepts messages from a fixed set of parties $\mathcal{P}$.

The ledger functionality $\mathcal{L}$ is initiated by the environment $\mathcal{E}$ via the following steps: (1) $\mathcal{E}$ instructs the ledger functionality to generate public parameter of the signature scheme pp; (2) $\mathcal{E}$ instructs every party $P \in \mathcal{P}$ to generate a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ and submit the public key pk_P to the ledger via the message $(\text{register}, \mathsf{pk}_P)$; (3) sets the initial state of the ledger meaning that it initialize a set $\mathtt{TX}$ defining all published transactions.

Once initialized, the state of $\mathcal{L}$ is public and can be accessed by all parties of the protocol, the adversary $\mathcal{A}$, and the environment $\mathcal{E}$. Any party $P \in \mathcal{P}$ can at any time post a transaction on the ledger via the message $(\text{post}, \mathsf{tx})$. The ledger functionality waits for at most Δ rounds (the exact number of rounds is determined by the adversary). Thereafter, the ledger verifies the validity of the transaction and adds it to the transaction set $\mathtt{TX}$. The formal description of the ledger functionality follows.

Ideal Functionality $\mathcal{L}(\Delta, \Sigma)$

The functionality accepts messages from all parties that are in the set $\mathcal{P}$ and maintains a $\mathtt{PKI}$ for those parties. The functionality maintains the set of all accepted transactions $\mathtt{TX}$ and all unspent transaction outputs $\mathtt{UTXO}$. The set $\mathcal{V}$ defines valid output conditions.

Initialize public keys: Upon $(\text{register}, \mathsf{pk}_P) \overset{\tau_0}{\hookleftarrow} P$ and it is the first time P sends a registration message, add (pk_P, P) to $\mathtt{PKI}$.

Post transaction: Upon $(\text{post}, \mathsf{tx}) \overset{\tau_0}{\hookleftarrow} P$, check that $|\mathtt{PKI}| = |\mathcal{P}|$. If not, drop the message, else

wait until round $\tau_1 \leq \tau_0 + \Delta$ (the exact value of τ_1 is determined by the adversary). Then check if:

1. The id is unique, i.e. for all $(t, \mathsf{tx}') \in \mathtt{TX}$, $\mathsf{tx}'.\mathsf{txid} \neq \mathsf{tx}.\mathsf{txid}$.
2. All the inputs are unspent and the witness satisfies all the output conditions, i.e. for each $(\mathsf{tid}, i) \in \mathsf{tx}.\mathsf{input}$, there exists $(t, \mathsf{tid}, i, \theta) \in \mathtt{UTXO}$ and $\theta.\varphi(\mathsf{tx}, t, \tau_1) = 1$.
3. All outputs are valid, i.e. for each $\theta \in \mathsf{tx}.\mathsf{output}$ it holds that $\theta.\mathsf{cash} > 0$ and $\theta.\varphi \in \mathcal{V}$.
4. The value of the outputs is not larger than the value of the inputs. More formally, let $I := \{\mathtt{utxo} := (t,\ \mathsf{tid},\ i,\ \theta) \mid \mathtt{utxo} \in \mathtt{UTXO} \wedge (\mathsf{tid},\ i) \in \mathsf{tx}.\mathsf{input}\}$, then it must hold that $\sum_{\theta' \in \mathsf{tx}.\mathsf{output}} \theta'.\mathsf{cash} \leq \sum_{\mathtt{utxo} \in I} \mathtt{utxo}.\theta.\mathsf{cash}$
5. The absolute time-lock of the transaction has expired, i.e. it must hold that $\mathsf{tx}.\mathsf{TimeLock} \leq \mathsf{now}$.

If all the above checks return true, add (τ_1, tx) to $\mathtt{TX}$, remove the spent outputs from $\mathtt{UTXO}$, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \setminus I$ and add the outputs of tx to $\mathtt{UTXO}$, i.e., $\mathtt{UTXO} := \mathtt{UTXO} \cup \{(\tau_1, \mathsf{tx}.\mathsf{txid}, i, \theta_i)\}_{i \in [n]}$ for $(\theta_1, \ldots, \theta_n) := \mathsf{tx}.\mathsf{output}$. Else, ignore the message.

Let us emphasize that our ledger functionality is fairly simplified. In reality, parties can join and leave the blockchain system dynamically. Moreover, we completely abstract from the fact that transactions are published in blocks which are proposed by parties and the adversary. Those and other features are captured by prior works, such as [BMTZ17], that provide a more accurate formalization of the Bitcoin ledger in the UC framework [Can01]. However, interaction with such ledger functionality is fairly complex. To increase the readability of our channel protocols and ideal functionality, which is the main focus of our work, we decided on this simpler ledger.

The GUC-security definition. Let π be a protocol with access to the global ledger $\mathcal{L}(\Delta, \Sigma)$, the global clock $\widehat{\mathcal{F}}_{clock}$ and ideal functionalities $\mathcal{H}_1, \ldots, \mathcal{H}_m$. The output of an environment $\mathcal{E}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input 1^λ and auxiliary input z is denoted as

$$EXEC_{\pi,\mathcal{A},\mathcal{E}}^{\mathcal{L}(\Delta,\Sigma),\widehat{\mathcal{F}}_{clock},\mathcal{H}_1,\ldots,\mathcal{H}_m}(\lambda, z).$$

Let $\phi_{\mathcal{F}}$ be the ideal protocol for an ideal functionality $\mathcal{F}$ with access to the global ledger $\mathcal{L}(\Delta, \Sigma)$ and the global clock $\widehat{\mathcal{F}}_{clock}$. This means that $\phi_{\mathcal{F}}$ is a trivial protocol in which the parties simply forward their inputs to the ideal functionality $\mathcal{F}$. The output of an environment $\mathcal{E}$ interacting with a protocol $\phi_{\mathcal{F}}$ and a adversary $\mathcal{S}$ (sometimes also call *simulator*) on input 1^λ and auxiliary input z is denoted as

$$EXEC_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}^{\mathcal{L}(\Delta,\Sigma),\widehat{\mathcal{F}}_{clock}}(\lambda, z).$$

We are now ready to state our main security definition which, informally, says that if a protocol π UC-realizes an ideal functionality $\mathcal{F}$, then any attack that can be carried out against the real-world protocol π can also be carried out against the ideal protocol $\phi_{\mathcal{F}}$.

Definition 17. A protocol π working in a $(\mathcal{H}_1, \ldots, \mathcal{H}_m)$-hybrid model *UC-realizes an ideal functionality $\mathcal{F}$ with respect to a global ledger $\mathcal{L} := \mathcal{L}(\Delta, \Sigma)$* and a global clock $\widehat{\mathcal{F}}_{clock}$

if for every adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that we have

$$\left\{EXEC^{\mathcal{L},\widehat{\mathcal{F}}_{clock},\mathcal{H}_1,\ldots,\mathcal{H}_m}_{\pi,\mathcal{A},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{EXEC^{\mathcal{L},\widehat{\mathcal{F}}_{clock}}_{\phi_{\mathcal{F}},\mathcal{S},\mathcal{E}}(\lambda,z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

(where "$\overset{c}{\approx}$" denotes computational indistinguishability of distribution ensembles, see, e.g., [Gol06]).

To simplify exposition, we omit the session identifiers *sid* and the sub-session identifiers *ssid*. Instead, we will use expressions like "message m is a reply to message m'". We believe that this approach improves readability.

E.2 Adaptor Signatures

Adaptor signatures have been introduced and used in the cryptocurrency community for some time, but have been formalized for the first time in [AEE+21]. These signatures not only allow for authentication as normal signature schemes do but also reveal a secret value upon publishing. Here we recall the definition of an adaptor signature scheme from [AEE+21]. In a nutshell, an adaptor signature is generated in two phases. First, a pre-signature is computed w.r.t. some statement Y of a hard relation R e.g. $Y = g^y$ where g is the generator of the group $\mathbb{G}$ in which computing the discrete logarithm is hard. We define L_R to be the associated language for R defined as $L_R := \{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. This pre-signature can be adapted to a full signature given a witness y for the statement Y, i.e. $(Y, y) \in R$. Furthermore, given the pre-signature and the adapted full signature one can extract a witness y. We now recall the definition for adaptor signature schemes from [AEE+21].

Definition 18 (Adaptor Signature Scheme)**.** An adaptor signature scheme wrt. a hard relation R and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ consists of four algorithms $\Xi_{R.\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ defined as:

$\mathsf{pSign}_{\mathsf{sk}}(m, Y)$**:** is a PPT algorithm that on input a secret key sk, message $m \in \{0,1\}^*$ and statement $Y \in L_R$, outputs a pre-signature $\tilde{\sigma}$.

$\mathsf{pVrfy}_{\mathsf{pk}}(m, Y; \tilde{\sigma})$**:** is a DPT algorithm that on input a public key pk, message $m \in \{0,1\}^*$, statement $Y \in L_R$ and pre-signature $\tilde{\sigma}$, outputs a bit b.

$\mathsf{Adapt}(\tilde{\sigma}, y)$**:** is a DPT algorithm that on input a pre-signature $\tilde{\sigma}$ and witness y, outputs a signature σ.

$\mathsf{Ext}(\sigma, \tilde{\sigma}, Y)$**:** is a DPT algorithm that on input a signature σ, pre-signature $\tilde{\sigma}$ and statement $Y \in L_R$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.

We now briefly recall the properties that an adaptor signature scheme must satisfy and refer the reader to [AEE+21] for the formal definitions.

Correctness. An adaptor signature should not only satisfy the standard signature correctness, but it must also satisfy *pre-signature correctness*. This property guarantees that if a pre-signature is generated honestly (wrt. a statement $Y \in L_R$), it can be *adapted* into a valid signature such that a witness for Y can be extracted.

Existential unforgeability under chosen message attack for adaptor signatures. Unforgeability for adaptor signatures is very similar to the normal definition of existential unforgeability under chosen message attacks for digital signatures, but it additionally requires that producing a forged signature σ for a message m is hard even if the adversary is given a pre-signature on the challenge message m w.r.t. a random statement $Y \in L_R$.

Pre-signature adaptability. Intuitively it is required that any *valid* pre-signature w.r.t. Y (even when produced by a malicious signer) can be completed into a valid signature using the witness y where $(Y, y) \in R$.

Witness extractability. In a nutshell, this property states that given a valid signature / pre-signature pair $(\sigma, \tilde{\sigma})$ for a message m with respect to a statement Y, one can extract the corresponding witness y.

E.3 Additional material to ledger channels

E.3.1 Ledger channels

For completeness, we recall the ledger channel ideal functionality $\mathcal{F}_L$ from [AEE+21]. We then show that we cannot use this ideal functionality in a black-box way and instead, we introduce a *wrapped ledger channel functionality* $\mathcal{F}_{preL}$. Finally, we present a protocol Π_{preL} that realizes $\mathcal{F}_{preL}$.

E.3.2 Ledger Channel Functionality

We now recall the ideal functionality for ledger channels $\mathcal{F}_L(T_p, k)$ from [AEE+21]. This functionality is parameterized by $T_p \in \mathbb{N}$ that upper bounds the number of consecutive off-chain communication rounds between parties and a parameter $k \in \mathbb{N}$ that defined the number of ways a channel can be closed (i.e., number of commit transactions per update).

Following [AEE+21], the pseudocode presented below excludes several checks that one would expect the functionality to make. We formalize all the missing checks in form of a functionality wrapper in Appendix E.5. Moreover, in order to simplify the notation in the functionality description, we write $m \overset{t}{\hookrightarrow} P$ as a short hand form for "send the message m to party P in round t." and $m \overset{t}{\hookleftarrow} P$ for "receive a message m from party P in round t".

Ideal Functionality $\mathcal{F}_L(T_p, k)$

We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Create

Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}_P) \xleftarrow{\tau_0} P$, let $\mathcal{S}$ define $T_1 \leq T$ and:

Both agreed: If already received $(\texttt{CREATE}, \gamma, \mathsf{tid}_Q) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_1$, wait if in round $\tau_1 \leq \tau + \Delta + T_1$ a transaction tx, with $\mathsf{tx.input} = (\mathsf{tid}_P, \mathsf{tid}_Q)$ and $\mathsf{tx.output} = (\gamma.\mathsf{cash}, \varphi)$, appears on the ledger $\mathcal{L}$. If yes, set $\Gamma(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$. Else stop.

Wait for Q: Else store the message and stop.

Update

Upon $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xleftarrow{\tau_0} P$, let $\mathcal{S}$ define $T_1, T_2 \leq T$, parse $(\gamma, \mathsf{tx}) := \Gamma(\mathsf{id})$ and proceed as follows:

1. In round $\tau_1 \leq \tau_0 + T_p$, let $\mathcal{S}$ set $|\mathsf{tid}| = k$. Then $(\texttt{UPDATE–REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{SETUP}, \mathsf{id}, \mathsf{tid}) \xhookrightarrow{\tau_1} P$.
2. If $(\texttt{SETUP–OK}, \mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_2 + T_1} Q$. Else stop.
3. If $(\texttt{UPDATE–OK}, \mathsf{id}) \xleftarrow{\tau_2 + T_1} Q$, then $(\texttt{UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_2 + 2T_1} P$. Else distinguish:
   - If Q honest or if instructed by $\mathcal{S}$, stop (update rejected).
   - Else execute $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
4. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_2 + 2T_1} P$, $(\texttt{REVOKE–REQ}, \mathsf{id}) \xhookrightarrow{\tau_2 + 2T_1 + T_2} Q$. Else execute $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
5. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_2 + 2T_1 + T_2} Q$, set $\gamma.\mathsf{st} = \theta$ and $\Gamma(\mathsf{id}) := (\gamma, \mathsf{tx})$. Then $(\texttt{UPDATED}, \mathsf{id}, \theta) \xhookrightarrow{\tau_2 + 2T_1 + 2T_2} \gamma.\mathsf{users}$ and stop. Else distinguish:
   - If Q honest, execute $\texttt{L–ForceClose}(\mathsf{id})$ and stop.
   - If Q corrupt, and wait for Δ rounds. If tx still unspent, then set $\theta_{old} := \gamma.\mathsf{st}$, $\gamma.\mathsf{st} := \{\theta_{old}, \theta\}$ and $\Gamma(\mathsf{id}) := (\gamma, \mathsf{tx})$. Execute $\texttt{L–ForceClose}(\mathsf{id})$ and stop.

Close

Upon $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_0} P$, let $\mathcal{S}$ define $T_1 \leq T$ and distinguish:

Both agreed: If you received $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_1$, let $(\gamma, \mathsf{tx}) := \Gamma(\mathsf{id})$ and distinguish:

- If in round $\tau_1 \leq \tau + T_1 + \Delta$ a transaction tx', with $\mathsf{tx}'.\mathsf{output} = \gamma.\mathsf{st}$ and $\mathsf{tx}'.\mathsf{input} = \mathsf{tx}.\mathsf{txid}$, appears on $\mathcal{L}$, set $\Gamma(\mathsf{id}) := (\bot, \mathsf{tx})$, $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$ and stop.
- If tx is still unspent in round $\tau + T_1 + \Delta$, output $(\texttt{ERROR}) \xhookrightarrow{\tau+T_1+\Delta} \gamma.\mathsf{users}$ and stop.

Wait for Q: Else wait for at most T_1 rounds to receive $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau \leq \tau_0 + T_1} Q$ (in that case option "Both agreed" is executed). If such message is not received, execute $\texttt{L–ForceClose}(\mathsf{id})$ in round $\tau_0 + T_1$.

Punish (executed at the end of every round τ_0)

For each $(\gamma, \mathsf{tx}) \in \Gamma$ check if $\mathcal{L}$ contains tx' with $\mathsf{tx}'.\mathsf{input} = \mathsf{tx}.\mathsf{txid}$. If yes, then distinguish:

Punish: For $P \in \gamma.\mathsf{users}$ honest, the following must hold: in round $\tau_1 \leq \tau_0 + \Delta$, a transaction tx'' with $\mathsf{tx}''.\mathsf{input} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{output} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{\mathsf{pk}_P})$ appears on $\mathcal{L}$. Then send $(\texttt{PUNISHED}, \mathsf{id}) \xhookrightarrow{\tau_1} P$, set $\Gamma(\mathsf{id}) := \bot$ and stop.

Close: Either $\Gamma(\mathsf{id}) = (\bot, \mathsf{tx})$ before round $\tau_0 + \Delta$ (channels was peacefully closed) or in round $\tau_1 \leq \tau_0 + 2\Delta$ a transaction tx'', with $\mathsf{tx}''.\mathsf{output} \in \gamma.\mathsf{st}$ and $\mathsf{tx}''.\mathsf{input} = \mathsf{tx}'.\mathsf{txid}$, appears on $\mathcal{L}$ (channel is forcefully closed). In the latter case, set $\Gamma(\mathsf{id}) := (\bot, \mathsf{tx})$ and $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{\tau_1} \gamma.\mathsf{users}$.

Error: Otherwise $(\texttt{ERROR}) \xhookrightarrow{\tau_0+2\Delta} \gamma.\mathsf{users}$.

Subprocedure $\texttt{L–ForceClose}(\mathsf{id})$

Let τ_0 be the current round and $(\gamma, \mathsf{tx}) := \Gamma(\mathsf{id})$. If within Δ rounds tx is still an unspent transaction on $\mathcal{L}$, then $(\texttt{ERROR}) \xhookrightarrow{\tau_0+\Delta} \gamma.\mathsf{users}$ and stop. Else, latest in round $\tau_0 + 3\Delta$, $m \in \{\texttt{CLOSED}, \texttt{PUNISHED}, \texttt{ERROR}\}$ is output via Punish.

E.3.3 Wrapped ledger channel functionality

For technical reasons, we cannot use the ledger channel functionality $\mathcal{F}_L(T_p, k)$ for building virtual channels in a black-box way. The main problem comes from the offloading feature of virtual channels. In order to overcome these issues, we present $\mathcal{F}_{preL}(T_p, k)$, an ideal functionality that extends $\mathcal{F}_L(T_p, k)$ to support the preparation of generalized channels ahead of time and later registration of such prepared generalized channels. Technically, the functionality extension is done by *wrapping* the original functionality. Before we present the functionality wrapper $\mathcal{F}_{preL}(T_p, k)$ formally, let us explain each of its parts on a high level.

Generalized channels. The functionality treats messages about standard generalized channels exactly as the functionality $\mathcal{F}_L(T_p, k)$ presented earlier in this section.

Creation. In order to pre-create a generalized channel γ, both end-users of the channel must send the message $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl})$ to the ideal functionality. Here $\mathsf{tx}^{\mathsf{f}}||i$ identifies the funding of the channel and $t_{ofl} \in \mathbb{N}$ represents the maximal number of rounds it should take to publish the channel funding transaction on-chain. If the functionality receives such a message from both parties within T_p rounds, it stores the channel γ, the funding identifier, and the waiting time to a special channel set Γ_{pre}, and informs both parties about the successful pre-creation.

Update. The update process works similarly as for standard ledger channels with one difference. If the update process fails at some point (e.g., one of the parties does not revoke), the functionality does not call `L–ForceClose` since there is no ledger channel to be forcefully closed. Instead, it calls a subprocedure called `Wait–if–Register` which add a flag "`in–dispute`" to the channel and waits for at most t_{ofl} rounds if the prepared channel is turned into a standard generalized channel (i.e., the corresponding funding transaction is added to the blockchain). If not, then it adds the new channel state back into the set of prepared (not yet full-fledged) channel states.

Register. The ideal functionality constantly monitors the ledger. Once the funding transaction of one of the channels in preparation appears on-chain, the functionality moves the information about the channel from the channel space Γ_{pre} to the channel space Γ. Moreover, if the channel in preparation was marked as "`in–dispute`", then it immediately calls `L–ForceClose`.

The formal functionality description on the functionality wrapper $\mathcal{F}_{preL}(T_p, k)$ follows. Again, for the sake of readability, we exclude several natural checks from the functionality description. These checks are formalized in Appendix E.5.

Wrapped Ledger Channel Functionality $\mathcal{F}_{preL}(T_p, k)$

We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Ledger Channels

Upon receiving a `CREATE`, `UPDATE`, `SETUP–OK`, `UPDATE–OK`, `REVOKE` or `CLOSE` message, then behave exactly as the functionality $\mathcal{F}_L(T_p, k)$.

Pre-Create

Upon $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}) \overset{\tau_0}{\hookleftarrow} P$, let $\mathcal{S}$ define $T_1 \leq T$ and:

Both agreed: If already received $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}) \xleftarrow{\tau} Q$, where $\tau_0 - \tau \leq T_1$, check that $\mathsf{tx}^{\mathsf{f}}.\mathsf{output}[i].\mathsf{cash} = \gamma.\mathsf{cash}$. If yes, set $\Gamma_{pre}(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, t_{ofl})$ and $(\texttt{PRE–CREATED}, \gamma.\mathsf{id}) \xhookrightarrow{\tau_0} \gamma.\mathsf{users}$. Else stop.

Wait for Q: Else store the message and stop.

Pre-Update

Upon $(\texttt{PRE–UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xleftarrow{\tau_0} P$, let $\mathcal{S}$ define $T_1, T_2 \leq T$, parse $(\gamma, \mathsf{tx}^{\mathsf{f}}, t) := \Gamma_{pre}(\mathsf{id})$ and proceed as follows:

1. In round $\tau_1 \leq \tau_0 + T_p$, let $\mathcal{S}$ set $|\mathsf{tid}| = k$. Then $(\texttt{PRE–UPDATE–REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xhookrightarrow{\tau_1} Q$ and $(\texttt{PRE–SETUP}, \mathsf{id}, \mathsf{tid}) \xhookrightarrow{\tau_1} P$.
2. If $(\texttt{PRE–SETUP–OK}, \mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1 + t_{\mathsf{stp}}} P$, then $(\texttt{PRE–SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_2 + T_1} Q$. Else stop.
3. In round $\tau_2 + T_1$ distinguish:
   - If $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \xleftarrow{\tau_2 + T_1} Q$, then $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_2 + 2T_1} P$.
   - If not and Q honest or if instructed by $\mathcal{S}$, $(\texttt{PRE–UPDATE–REJECT}, \mathsf{id}) \xhookrightarrow{\tau_2 + 2T_1} P$.
   - Else execute $\texttt{Wait–if–Register}(\mathsf{id})$ and stop.
4. If $(\texttt{PRE–REVOKE}, \mathsf{id}) \xleftarrow{\tau_2 + 2T_1} P$, $(\texttt{PRE–REVOKE–REQ}, \mathsf{id}) \xhookrightarrow{\tau_2 + 2T_1 + T_2} Q$. Else execute $\texttt{Wait–if–Register}(\mathsf{id})$ and stop.
5. If $(\texttt{PRE–REVOKE}, \mathsf{id}) \xleftarrow{\tau_2 + 2T_1 + T_2} Q$, set $\gamma.\mathsf{st} = \theta$ and $\Gamma_V(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}})$. Then $(\texttt{PRE–UPDATED}, \mathsf{id}, \theta) \xhookrightarrow{\tau_2 + 2T_1 + 2T_2} \gamma.\mathsf{users}$ and stop. Else $\texttt{Wait–if–Register}(\mathsf{id})$ and stop.

Register – executed in every round

Let t_0 be the current round. For every $(\gamma, \mathsf{tx}^{\mathsf{f}}) \in \Gamma_{pre}$ check if tx^{f} appears on the ledger $\mathcal{L}$. If yes, then $\Gamma_{pre}(\gamma.\mathsf{id}) = \bot$ and $\Gamma(\gamma.\mathsf{id}) = (\gamma, \mathsf{tx}^{\mathsf{f}})$.

Subprocedure $\texttt{Wait–if–Register}(\mathsf{id})$

Let τ_0 be the current round and $(\gamma, \mathsf{tx}^{\mathsf{f}}, t_{ofl}) := \Gamma_{pre}(\mathsf{id})$.

1. Set $\Gamma_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, t_{ofl}, \texttt{in–dispute})$.
2. Wait for t_{ofl} rounds. If after this time, $\Gamma_{pre}(\mathsf{id}) \neq \bot$, then set $\theta_{old} := \gamma.\mathsf{st}$, $\gamma.\mathsf{st} := \{\theta_{old}, \theta\}$ and $\Gamma_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, t_{ofl}, \texttt{in–dispute})$.

E.3.4 Realizing the wrapped functionality

While [AEE+21] presents a protocol Π_L that realizes the ideal functionality $\mathcal{F}_L$, it does not say anything about our wrapped functionality $\mathcal{F}_{preL}$. In order to have such protocol, we design a *protocol wrapper* around the protocol Π_L and prove that such wrapped protocol, which we denote Π_{preL} realizes the ideal functionality $\mathcal{F}_{preL}$. Note that, just like Π_L, Π_{preL} uses the adaptor signature primitive that we recalled in Appendix E.2.

Let us stress that the protocol wrapper very closely follows the protocol for ledger channels. Below we stress the main difference and thereafter we formally define the protocol for completeness.

Pre-Create. The only difference between the pre-create and create is that in pre-create tx^{f} is neither generated by the parties nor posted on the ledger and is given as an input from the environment. Intuitively this is a funding transaction that might be posted in the future. Hence such channels are called pre-created or prepared channels.
Pre-Update. During the pre-update procedure, parties update the state of the pre-created channel as in normal ledger channels, but parties cannot directly force-close the channel since the funding transaction is not posted on the ledger yet. Hence in case of dispute parties first have to post this transaction on the ledger this is captured in calls to `Wait–if–Register` sub-procedure.

Register. This is a new procedure in order to capture the situation during which the funding transaction of a pre-created channel is posted on the ledger. In this case, the pre-created channel is transformed into a normal ledger channel and is added to the list of ledger channels. Furthermore, if this channel was in dispute, it is directly force closed.

To sum up, parties upon receiving one of the `PRE–UPDATE`, `PRE–SETUP–OK`, `PRE–UPDATE–OK` or `PRE–REVOKE` messages, behave as in the protocol Π_L with the following changes:

- Use the channel space Γ^P_{pre} instead of Γ^P.
- Add t_{ofl} rounds to the absolute time lock of new $\mathtt{TX}_{\mathsf{c}}$.
- Replace calls to $\mathtt{L–ForceClose}^P$ by calls to $\mathtt{Wait–if–Register}^P$ which marks a channel to be in dispute.
- In case the reacting party peacefully rejects the update, output `PRE–UPDATE–REJECT` before you stop.
- When the protocol instructs you to output a m-message, where $m \in \{\mathtt{UPDATE–REQ}, \mathtt{SETUP}, \mathtt{SETUP–OK}, \mathtt{UPDATE–OK}, \mathtt{REVOKE–REQ}, \mathtt{UPDATED}\}$, then output PRE–m.

We are now prepared to present the formal description of the protocol. As for the ideal functionalities, we exclude several natural checks that parties have to make. We present all these checks in the form of a protocol wrapper in Appendix E.5.

Wrapped Ledger Channel Protocol Π_{preL}

Below, we abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$.

Ledger channels

Upon receiving a CREATE, UPDATE, SETUP–OK, UPDATE–OK, REVOKE or CLOSE message, then behave exactly as in the protocol Π_L.

Pre-Create

Party P upon $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}) \xleftarrow{t_0} \mathcal{E}$:

1. If $\mathsf{tx}^{\mathsf{f}}.\mathsf{output}[i].\mathsf{cash} \neq \gamma.\mathsf{cash}$, then ignore the message.
2. Set $\mathsf{id} := \gamma.\mathsf{id}$, generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{createInfo}, \mathsf{id}, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}, R_P, Y_P) \xhookrightarrow{t_0} Q$.
3. If $(\text{createInfo}, \mathsf{id}, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}, R_Q, Y_Q) \xhookleftarrow{t_0+1} Q$, create:

$$[\mathtt{TX}_{\mathsf{c}}] := \mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_P, I_Q, 0)$$
$$[\mathtt{TX}_{\mathsf{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \gamma.\mathsf{st})$$

   for $I_P := (\mathsf{pk}_P, R_P, Y_P)$, $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$. Else stop.
4. Compute $s_{\mathsf{c}}^P \leftarrow \mathsf{pSign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{c}}], Y_Q)$, $s_{\mathsf{s}}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{s}}])$ and send $(\text{createCom}, \mathsf{id}, s_{\mathsf{c}}^P, s_{\mathsf{s}}^P) \xhookrightarrow{t_0+1} Q$.
5. If $(\text{createCom}, \mathsf{id}, s_{\mathsf{c}}^Q, s_{\mathsf{s}}^Q) \xhookleftarrow{t_0+2} Q$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_{\mathsf{c}}], Y_P; s_{\mathsf{c}}^Q) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^Q) = 1$, set

$$\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^Q, y_P)\})$$
$$\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^P, s_{\mathsf{s}}^Q\})$$
$$\Gamma_{pre}^P(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathsf{c}}, r_P, R_Q, Y_Q, s_{\mathsf{c}}^P), \mathtt{TX}_{\mathsf{s}}, t_{ofl}).$$

   and send $(\texttt{PRE–CREATED}, \mathsf{id}) \xhookrightarrow{t_0+2} \mathcal{E}$.

Pre-Update

Party P upon $(\texttt{PRE–UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookleftarrow{t_0} \mathcal{E}$

1. Generate $(R_P, r_P) \leftarrow \mathsf{GenR}$, $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send the message $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookrightarrow{t_0} Q$.

Party Q upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_P, Y_P) \xhookleftarrow{\tau_0} P$

2. Generate $(R_Q, r_Q) \leftarrow \mathsf{GenR}$ and $(Y_Q, y_Q) \leftarrow \mathsf{GenR}$.
3. Extract tx^{f} and t_{ofl} from $\Gamma_{pre}^P(\mathsf{id})$.

4. Set $t_{\mathsf{lock}} := \tau_0 + t_{\mathsf{stp}} + 4 + \Delta + t_{ofl}$ and

$$[\mathtt{TX}_\mathsf{c}] := \mathtt{GenCommit}([\mathsf{tx}^\mathsf{f}], I_P, I_Q, t_{\mathsf{lock}})$$
$$[\mathtt{TX}_\mathsf{s}] := \mathtt{GenSplit}([\mathtt{TX}_\mathsf{c}].\mathsf{txid} \| 1, \theta)$$

where $I_P := (\mathsf{pk}_P, R_P, Y_P)$, $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$.

5. Sign $s_\mathsf{s}^Q \leftarrow \mathsf{Sign}_{\mathsf{sk}_Q}([\mathtt{TX}_\mathsf{s}])$, send $(\mathsf{updateInfo}, \mathsf{id}, R_Q, Y_Q, s_\mathsf{s}^Q) \overset{\tau_0}{\hookrightarrow} P$, $(\mathtt{PRE\text{–}UPDATE\text{–}REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathtt{TX}_\mathsf{s}.\mathsf{txid}) \overset{\tau_0+1}{\hookrightarrow} \mathcal{E}$.

Party P upon $(\mathsf{updateInfo}, \mathsf{id}, h_Q, Y_Q, s_\mathsf{s}^Q) \overset{t_0+2}{\hookleftarrow} Q$

6. Extract tx^f and t_{ofl} from $\Gamma_{pre}^Q(\mathsf{id})$.
7. Set $t_{\mathsf{lock}} := t_0 + t_{\mathsf{stp}} + 5 + \Delta + t_{ofl}$, and

$$[\mathtt{TX}_\mathsf{c}] := \mathtt{GenCommit}([\mathsf{tx}^\mathsf{f}], I_P, I_Q, t_{\mathsf{lock}})$$
$$[\mathtt{TX}_\mathsf{s}] := \mathtt{GenSplit}([\mathtt{TX}_\mathsf{c}].\mathsf{txid} \| 1, \theta),$$

for $I_P := (\mathsf{pk}_P, R_P, Y_P)$ and $I_Q := (\mathsf{pk}_Q, R_Q, Y_Q)$. If it holds that $\mathsf{Vrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_\mathsf{s}]; s_\mathsf{s}^Q) = 1$, $(\mathtt{PRE\text{–}SETUP}, \mathsf{id}, \mathtt{TX}_\mathsf{s}.\mathsf{txid}) \overset{t_0+2}{\hookrightarrow} \mathcal{E}$. Else stop.

8. If $(\mathtt{PRE\text{–}SETUP\text{–}OK}, \mathsf{id}) \overset{t_1 \le t_0+2+t_{\mathsf{stp}}}{\hookleftarrow} \mathcal{E}$, compute the values $s_\mathsf{c}^P \leftarrow \mathsf{pSign}_{\mathsf{sk}_P}([\mathtt{TX}_\mathsf{c}], Y_Q)$, $s_\mathsf{s}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_\mathsf{s}])$ and send the message $(\mathsf{updateComP}, \mathsf{id}, s_\mathsf{c}^P, s_\mathsf{s}^P) \overset{t_1}{\hookrightarrow} Q$. Else stop.

Party Q

9. If $(\mathsf{updateComP}, \mathsf{id}, s_\mathsf{c}^P, s_\mathsf{s}^P) \overset{\tau_1 \le \tau_0+2+t_{\mathsf{stp}}}{\hookleftarrow} P$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_P}([\mathtt{TX}_\mathsf{c}], Y_Q; s_\mathsf{c}^P) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_P}([\mathtt{TX}_\mathsf{s}]; s_\mathsf{s}^P) = 1$, output $(\mathtt{PRE\text{–}SETUP\text{–}OK}, \mathsf{id}) \overset{\tau_1}{\hookrightarrow} \mathcal{E}$. Else stop.
10. If $(\mathtt{PRE\text{–}UPDATE\text{–}OK}, \mathsf{id}) \overset{\tau_1}{\hookleftarrow} \mathcal{E}$, pre-sign $s_\mathsf{c}^Q \leftarrow \mathsf{pSign}([\mathtt{TX}_\mathsf{c}], Y_P)$ and send $(\mathsf{updateComQ}, \mathsf{id}, s_\mathsf{c}^Q) \overset{\tau_1}{\hookrightarrow} P$. Else send the message $(\mathsf{updateNotOk}, \mathsf{id}, r_Q) \overset{\tau_1}{\hookrightarrow} P$ and stop.

Party P

11. In round $t_1 + 2$ distinguish the following cases:
    - If $(\mathsf{updateComQ}, \mathsf{id}, s_\mathsf{c}^Q) \overset{t_1+2}{\hookleftarrow} Q$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_Q}([\mathtt{TX}_\mathsf{c}], Y_P; s_\mathsf{c}^Q) = 1$, output $(\mathtt{PRE\text{–}UPDATE\text{–}OK}, \mathsf{id}) \overset{t_1+2}{\hookrightarrow} \mathcal{E}$.
    - If $(\mathsf{updateNotOk}, \mathsf{id}, r_Q) \overset{t_1+2}{\hookleftarrow} Q$, s.t. $(R_Q, r_Q) \in R$, add $\Theta^P(\mathsf{id}) := \Theta^P(\mathsf{id}) \cup ([\mathtt{TX}_\mathsf{c}], r_Q, Y_Q, s_\mathsf{c}^P)$, output the message $(\mathtt{PRE\text{–}UPDATE\text{–}REJECT}) \overset{t_1+2}{\hookrightarrow} \mathcal{E}$ and stop.
    - Else, execute the procedure $\mathtt{Wait\text{–}if\text{–}Register}^P(\mathsf{id})$ and stop.

12. If $(\texttt{PRE–REVOKE}, \mathsf{id}) \xleftarrow{t_1+2} \mathcal{E}$, parse $\Gamma^P_{pre}(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\mathtt{TX}}_{\mathtt{c}}, \bar{r}_P, \bar{R}_Q, \bar{Y}_Q, \bar{s}^P_{\mathtt{Com}}), \overline{\mathsf{tx}}^{\mathsf{s}})$ and update the channel space as $\Gamma^P_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathtt{c}}, r_P, R_Q, Y_Q, s^P_{\mathtt{c}}), \mathtt{TX}_{\mathtt{s}})$, for $\mathtt{TX}_{\mathtt{s}} := ([\mathtt{TX}_{\mathtt{s}}], \{s^P_{\mathtt{s}}, s^Q_{\mathtt{s}}\})$ and $\mathtt{TX}_{\mathtt{c}} := ([\mathtt{TX}_{\mathtt{c}}], \{\mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathtt{c}}]), \mathsf{Adapt}(s^Q_{\mathtt{c}}, y_P)\})$., and send $(\text{revokeP}, \mathsf{id}, \bar{r}_P) \xrightarrow{t_1+2} Q$. Else, execute $\texttt{Wait–if–Register}^P(\mathsf{id})$ and stop.

Party Q

13. Parse $\Gamma^Q_{pre}(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\mathtt{TX}}_{\mathtt{c}}, \bar{r}_Q, \bar{R}_P, \bar{Y}_P, \bar{s}^Q_{\mathtt{Com}}), \overline{\mathsf{tx}}^{\mathsf{s}})$. If $(\text{revokeP}, \mathsf{id}, \bar{r}_P) \xleftarrow{\tau_1+2} P$, s.t. $(\bar{R}_P, \bar{r}_P) \in R$, $(\texttt{PRE–REVOKE–REQ}, \mathsf{id}) \xrightarrow{\tau_1+2} \mathcal{E}$. Else execute $\texttt{Wait–if–Register}^Q(\mathsf{id})$ and stop.
14. If $(\texttt{PRE–REVOKE}, \mathsf{id}) \xleftarrow{\tau_1+2} \mathcal{E}$ as a reply, set

$$\Theta^Q(\mathsf{id}) := \Theta^Q(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathtt{c}}], \bar{r}_P, \bar{Y}_P, \bar{s}^Q_{\mathtt{Com}})$$
$$\Gamma^Q_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathtt{c}}, r_Q, R_P, Y_P, s^Q_{\mathtt{c}}), \mathtt{TX}_{\mathtt{s}}),$$

for $\mathtt{TX}_{\mathtt{s}} := ([\mathtt{TX}_{\mathtt{s}}], \{s^P_{\mathtt{s}}, s^Q_{\mathtt{s}}\})$, $\mathtt{TX}_{\mathtt{c}} := ([\mathtt{TX}_{\mathtt{c}}], \{\mathsf{Sign}_{\mathsf{sk}_Q}([\mathtt{TX}_{\mathtt{c}}]), \mathsf{Adapt}(s^P_{\mathtt{c}}, y_Q)\})$, and send $(\text{revokeQ}, \mathsf{id}, \bar{r}_Q) \xrightarrow{\tau_1+2} P$. In the next round $(\texttt{PRE–UPDATED}, \mathsf{id}) \xrightarrow{\tau_1+3} \mathcal{E}$ and stop. Else, in round $\tau_1 + 2$, execute $\texttt{Wait–if–Register}^Q(\mathsf{id})$ and stop.

Party P

15. If $(\text{revokeQ}, \mathsf{id}, \bar{r}_Q) \xleftarrow{t_1+4} Q$ s.t. $(\bar{R}_Q, \bar{r}_Q) \in R$, then set $\Theta^P(\mathsf{id}) := \Theta^P(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathtt{c}}], \bar{r}_Q, \bar{Y}_Q, \bar{s}^P_{\mathtt{Com}})$ and $(\texttt{PRE–UPDATED}, \mathsf{id}) \xrightarrow{t_1+4} \mathcal{E}$. Else execute $\texttt{Wait–if–Register}^P(\mathsf{id})$ and stop.

Register

Party P in every round t_0: For each $\mathsf{id} \in \{0,1\}^*$ s.t. $\Gamma^P_{pre}(\mathsf{id}) \neq \bot$:

1. Parse $\Gamma^P_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathtt{c}}, r_P, R_Q, Y_Q, s^P_{\mathtt{c}}), \mathtt{TX}_{\mathtt{s}}, t_{ofl}, x)$
2. If tx^{f} appeared on-chain in this round, then
   a) Set $\Gamma(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathtt{c}}, r_P, R_Q, Y_Q, s^P_{\mathtt{c}}), \mathtt{TX}_{\mathtt{s}})$.
   b) Set $\Gamma^P_{pre}(\mathsf{id}) := \bot$
   c) If $x = \texttt{in–dispute}$, then call $\texttt{L–ForceClose}^P(\mathsf{id})$.

Subprocedures

$\texttt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], (\mathsf{pk}_P, R_P, Y_P), (\mathsf{pk}_Q, R_Q, Y_Q), t)$:
Let $(c, \texttt{Multi–Sig}_{\mathsf{pk}_P,\mathsf{pk}_Q}) := \mathsf{tx}^{\mathsf{f}}.\mathsf{output}[1]$ and denote

$$\varphi_1 := \texttt{Multi–Sig}_{\mathsf{ToKey}(R_Q),\mathsf{ToKey}(Y_Q),\mathsf{pk}_P},$$
$$\varphi_2 := \texttt{Multi–Sig}_{\mathsf{ToKey}(R_P),\mathsf{ToKey}(Y_P),\mathsf{pk}_Q},$$
$$\varphi_3 := \texttt{CheckRelative}_{\Delta} \wedge \texttt{Multi–Sig}_{\mathsf{pk}_P,\mathsf{pk}_Q}.$$

Return $[\mathsf{tx}]$, where $\mathsf{tx}.\mathsf{input} = \mathsf{tx}^{\mathsf{f}}.\mathsf{txid}\|1$, $\mathsf{tx}.\mathsf{output} := (c, \varphi_1 \vee \varphi_2 \vee \varphi_3)$ and set $\mathsf{tx}.\mathsf{TimeLock}$ to t if $t >$ now and to 0 otherwise.

$\texttt{GenSplit}(\mathsf{tid}, \theta)$:
Return $[\mathsf{tx}]$, where $\mathsf{tx}.\mathsf{input} := \mathsf{tid}$ and $\mathsf{tx}.\mathsf{output} := \theta$.

$\texttt{Wait–if–Register}^{P}(\mathsf{id})$:
Let t_0 be the current round. Let $X := \Gamma^{P}_{pre}(\mathsf{id})$. Then set $\Gamma^{P}_{pre}(\mathsf{id}) := (X, \texttt{in–dispute})$.

Theorem 17. *Let Σ be a signature scheme that is existentially unforgeable against chosen message attacks, R a hard relation, and $\Xi_{R,\Sigma}$ a secure adaptor signature scheme. Then for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π_{preL} UC-realizes the ideal functionality $\mathcal{F}_{preL}(3, 1)$.*

E.4 Virtual Channels

In this section, we first describe and present the ideal functionality $\mathcal{F}_V$ that describes the ideal behavior of virtual channels. We then give a high-level description of our construction of virtual channels with validity, before giving the full formal protocol description of our virtual channel constructions.

E.4.1 Ideal functionality for virtual channels

$\mathcal{F}_V$ can be viewed as an extension of the ledger channel functionality $\mathcal{F}_L$ defined in [AEE+21] and here presented in Appendix E.3.1. The functionality $\mathcal{F}_V$ is parameterized by a parameter T which upper bounds the maximum number of off-chain communication rounds between two parties required for any of the operations in $\mathcal{F}_L$. The ideal functionality $\mathcal{F}_V$ communicates with the parties $\mathcal{P}$, the simulator $\mathcal{S}$ and the ledger $\mathcal{L}$ (see Appendix E.1). It maintains a channel space Γ where it stores all currently opened ledger channels (together with their funding transaction tx) and virtual channels. Before we define $\mathcal{F}_V$ formally, we describe it at a high level.

Messages related to ledger channels. For any message related to a ledger channel, $\mathcal{F}_V$ behaves as the functionality $\mathcal{F}_L$. That is, the corresponding code of $\mathcal{F}_L$ is executed when a message about a ledger channel γ is received. For the rest of this section, we discuss the behavior of $\mathcal{F}_V$ upon receiving a message about a virtual channel.

Create. The creation of a virtual channel is equivalent to synchronously updating two ledger channels. Therefore, if all parties, namely $\gamma.\mathsf{Alice}$, $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$, follow the

protocol, i.e., update their ledger channels correctly, a virtual channel is successfully created. This is captured in the "All agreed" case of the functionality. Hence, if all parties send the CREATE message, the functionality returns CREATED to γ.users, keeps the underlying ledger channels locked and adds the virtual channel to its channel space Γ.

On the other hand, the creation of the virtual channel fails if after some time at least one of the parties does not send CREATE to the functionality. There are three possible situations: (i), the update is peacefully rejected and parties simply abort the virtual channel creation, (ii) both channels are forcefully closed, in order to prevent a situation where one of the channels is updated and the other one is not, (iii) if γ.Ingrid has not published the old state of one of her channels to the ledger after Δ rounds, it forcefully closes the ledger channels using the new state i.e., where γ.Ingrid behaves maliciously and can publish both the old and new states, while γ.Alice or γ.Bob can only publish the new state.

Update. The update procedure for the virtual channel works in the same way as for ledger channels except in case of any disputes during the execution, the functionality calls V–ForceClose instead of L–ForceClose.

Offload. We consider two types of offloading depending on whether the virtual channel is with or without validity. In the first case, offloading is initiated by one of the γ.users before round γ.val, while for channels without validity, Ingrid can initiate the offloading at any time. Since offloading a virtual channel requires closure of the underlying subchannels, the functionality merely checks if either funding transaction of γ.subchan has been spent until round $T_1 + \Delta$. If not, the functionality outputs a message (ERROR). As in to [AEE+21], the ERROR message represents an impossible situation which should not happen as long as one of the parties is honest.

Close - channels without validity. Upon receiving (CLOSE, id) from all parties in γ.users within $T_1 \leq 6T$ rounds (where the exact value of T_1 is specified by $\mathcal{S}$), all parties have peacefully agreed on closing the virtual channel, which is indicated by the "All Agreed" case. In this case, the final balance of the parties is reflected on their underlying channels. When the update of Γ is completed, the ideal functionality sends CLOSED to all users. Due to the peaceful closure in this "All Agreed" case, the functionality defines property (E3).

If one of the (CLOSE, id) messages was not received within T_1 rounds ("Wait for others" case), the closing procedure fails. The following cases my happen: (i) the update procedure of an underlying ledger channel was aborted prematurely by γ.Alice or γ.Bob which would cause the virtual channel to be forcefully closed. (ii) γ.Ingrid refuses to revoke her state during the update of either one of the underlying ledger channels where the functionality waits Δ rounds and if γ.Ingrid has not published the old state to the ledger the functionality forcefully closes the ledger channels using the new state.

Close - channels with validity. This procedure starts in round γ.val $- (4\Delta + 7T)$ to have enough time to forcefully close the channel if necessary. If within $T_1 \leq 6T$ rounds

(where the exact value of T_1 is specified by $\mathcal{S}$) all $\gamma.\mathsf{users}$ agreed on closing the channel or if the simulator instructs the functionality to close the channel, the same steps as in the all agreed case for channels without validity are executed. Otherwise, after T_1 rounds, the functionality executes the forceful closure of the virtual channel.

Punish. The punishment procedure is executed at the end of each round. It checks for every virtual channel γ if any of $\gamma.\mathsf{subchan}$ has just been closed and distinguishes if the consequence of closure was offloading or punishment. If after T_1 rounds (T_1 is set by $\mathcal{S}$) two transactions tx_1 and tx_2 are published on the ledger, where tx_1 refunds the collateral $\gamma.\mathsf{cash} + \gamma.\mathsf{fee}$ to $\gamma.\mathsf{Ingrid}$ and tx_2 funds γ on-chain, then the virtual channel has been offloaded and the message (`OFFLOADED`) is sent to $\gamma.\mathsf{users}$. If after T_1 rounds, only one transaction tx is on the ledger, which assigns $\gamma.\mathsf{cash}$ coins to a single honest party P and spends the funding transaction of only one of $\gamma.\mathsf{subchan}$, the functionality sends (`PUNISHED`) to P. Otherwise, the functionality outputs (`ERROR`) to $\gamma.\mathsf{users}$.

Notation. In the functionality description, we use the notion of *rooted transactions* that we now explain (see Figure E.1 for a concrete example). UTXO-based blockchains can be viewed as a directed acyclic graph, where each node represents a transaction. Nodes corresponding to transactions tx_i and tx_j are connected with an edge if at least one of the outputs of tx_i is an input of tx_j, i.e, tx_i is (partially) funding tx_j. We denote the transitive reachability relation between nodes, which constitutes a partial order, as $\leq$. We say that a transaction tx is *rooted* in the set of transactions R if

1. $\forall \mathsf{tx}_i \leq \mathsf{tx}.\exists \mathsf{tx}_j \in R.\mathsf{tx}_j \leq \mathsf{tx}_i \vee \mathsf{tx}_i \leq \mathsf{tx}_j$,
2. $\forall \mathsf{tx}_i, \mathsf{tx}_j \in R.\mathsf{tx}_i \neq \mathsf{tx}_j, \mathsf{tx}_i \not\leq \mathsf{tx}_j$ and
3. $\mathsf{tx} \notin R$.

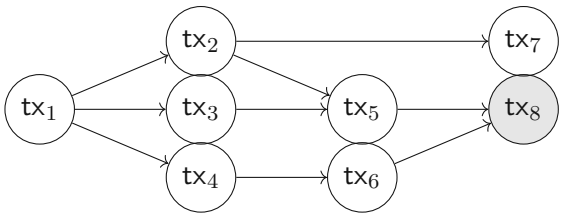

Figure E.1: The root sets of transaction tx_8 are $\{\mathsf{tx}_1\}$, $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4\}$, $\{\mathsf{tx}_5, \mathsf{tx}_6\}$, $\{\mathsf{tx}_4, \mathsf{tx}_5\}$ and $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_6\}$.

As in the case of ledger channel functionalities, the formal description of $\mathcal{F}_V$ excludes several checks that one would expect the functionality to make. These checks are formalized in form of a functionality wrapper in Appendix E.5.

Ideal Functionality $\mathcal{F}_V(T_p)$

Below we abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$, $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{users}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$.

For messages about ledger channels, behave as $\mathcal{F}_L(T_p, 1)$.

Create

Upon $(\texttt{CREATE}, \gamma) \overset{\tau}{\hookleftarrow} P$, let $\mathcal{S}$ define $T_1 \leq 8T$. If $P \in \gamma.\mathsf{users}$, then define a set S, where $S := \{\mathsf{id}_P\} := \gamma.\mathsf{subchan}(P)$, otherwise define S as $S := \{\mathsf{id}_P, \mathsf{id}_Q\} := \gamma.\mathsf{subchan}$. Lock all channels in S and distinguish:

All agreed: If you already received both $(\texttt{CREATE}, \gamma) \overset{\tau_1}{\hookleftarrow} Q_1$ and $(\texttt{CREATE}, \gamma) \overset{\tau_2}{\hookleftarrow} Q_2$, where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ and $\tau - T_1 \leq \tau_1 \leq \tau_2$, then in round $\tau_3 := \tau_1 + T_1$ proceed as:

1. Let $\mathcal{S}$ define $\theta_{\mathbf{A}}$ and $\theta_{\mathbf{B}}$ and set $(\mathsf{id}_A, \mathsf{id}_B) := \gamma.\mathsf{subchan}$.
2. Execute $\texttt{UpdateState}(\mathsf{id}_A, \theta_{\mathbf{A}})$, $\texttt{UpdateState}(\mathsf{id}_B, \theta_{\mathbf{B}})$, set $\Gamma(\gamma.\mathsf{id}) := \gamma$, send $(\texttt{CREATED}, \gamma) \overset{\tau_3}{\hookrightarrow} \gamma.\mathsf{users}$, stop.

Wait for others: Else wait for at most T_1 rounds to receive $(\texttt{CREATE}, \gamma) \overset{\tau_1 \leq \tau + T_1}{\longleftarrow} Q_1$ and $(\texttt{CREATE}, \gamma) \overset{\tau_2 \leq \tau + T_1}{\longleftarrow} Q_2$ where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ (in that case option "All agreed" is executed). If at least one of those messages does not arrive before round $\tau + T_1$, do the following. For all $\mathsf{id}_i \in S$, let $(\gamma_i, \mathsf{tx}_i) := \Gamma(\mathsf{id}_i)$ and distinguish the following cases:

- If $\mathcal{S}$ sends $(\text{peaceful–reject}, \mathsf{id}_i)$, unlock id_i and stop.
- If $\gamma.\mathsf{Ingrid}$ is honest or if instructed by $\mathcal{S}$, execute $\texttt{L–ForceClose}(\mathsf{id}_i)$ and stop.
- Otherwise wait for Δ rounds. If tx_i still unspent, then set $\theta_{old} := \gamma_i.\mathsf{st}$, $\gamma_i.\mathsf{st} := \{\theta_{old}, \theta\}$ and $\Gamma(\mathsf{id}_i) := (\gamma_i, \mathsf{tx}_i)$. Execute $\texttt{L–ForceClose}(\mathsf{id}_i)$ and stop.

Update

Upon $(\texttt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, where $P \in \gamma.\mathsf{users}$, behave as $\mathcal{F}_L(T_p, 1)$ yet replace the calls to $\texttt{L–ForceClose}$ in $\mathcal{F}_L(T_p, 1)$ with calls to $\texttt{V–ForceClose}$.

Offload

Upon $(\texttt{OFFLOAD}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, execute $\texttt{Offload}(\mathsf{id})$.

Close

Channels without validity:

Upon $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau} P$, where $\gamma(\mathsf{id}).\mathsf{val} = \bot$, let $\mathcal{S}$ define $T_1 \leq 6T_p$. If $P \in \gamma_i.\mathsf{users}$, define a set S, where $S := \{\mathsf{id}_P\} := \gamma_i.\mathsf{subchan}(P)$, else define S as $S := \{\mathsf{id}_P, \mathsf{id}_Q\} := \gamma_i.\mathsf{subchan}$ and distinguish:

All agreed: If you received both messages $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_1} Q_1$ and $(\texttt{CLOSE}, \mathsf{id}) \xleftarrow{\tau_2} Q_2$, where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ and $\tau - T_1 \leq \tau_1 \leq \tau_2$, then in round $\tau_3 := \tau_1 + T_1$ proceed as follows:

1. Let $\gamma := \Gamma(\mathsf{id})$, $(\mathsf{id}_A, \mathsf{id}_B) := \gamma.\mathsf{subchan}$.
2. Parse $\gamma.\mathsf{st} = \{(c_A, \texttt{One–Sig}_A), (c_B, \texttt{One–Sig}_B)\}$ and set

$$\theta_{\mathbf{A}} := ((c_A, \texttt{One–Sig}_A), (c_B + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_I)),$$
$$\theta_{\mathbf{B}} := ((c_A + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_I), (c_B, \texttt{One–Sig}_B)),$$

3. Unlock both subchannels and execute $\texttt{UpdateState}(\mathsf{id}_A, \theta_{\mathbf{A}})$ and $\texttt{UpdateState}(\mathsf{id}_B, \theta_{\mathbf{B}})$. Set $\Gamma(\mathsf{id}) := \bot$ and send $(\texttt{CLOSED}, \gamma) \xhookrightarrow{\tau_3} \gamma.\mathsf{users}$.

Wait for others: Else wait for at most T_1 rounds to receive $(\texttt{CLOSE}, \gamma) \xleftarrow{\tau_1 \leq \tau + T_1} Q_1$ and $(\texttt{CLOSE}, \gamma) \xleftarrow{\tau_2 \leq \tau + T_1} Q_2$ where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ (in that case option "All agreed" is executed). For all $\mathsf{id}_i \in S$ let $(\gamma_i, \mathsf{tx}_i) := \Gamma(\mathsf{id}_i)$, if such messages are not received until round $\tau + T_1$, set $\theta_{old} := \gamma'.\mathsf{st}$ and distinguish:

- If $\gamma.\mathsf{Ingrid}$ is honest or if instructed by $\mathcal{S}$, execute $\texttt{V–ForceClose}(\mathsf{id}_i)$ and stop.
- Else wait for Δ rounds. If tx_i still unspent, set $\gamma_i.\mathsf{st} := \{\theta_{old}, \theta\}$ and $\Gamma(\mathsf{id}_i) := (\gamma_i, \mathsf{tx}_i)$. Execute $\texttt{L–ForceClose}(\mathsf{id}_i)$ and stop.

Channels with validity:

For every $\gamma \in \Gamma$ s.t. $\gamma.\mathsf{val} \neq \bot$, in round $\tau_0 := \gamma.\mathsf{val} - (4\Delta + 7T_p)$ proceed as follows: let $\mathcal{S}$ set $T_1 \leq 6T_p$ and distinguish:

Peaceful close: If all parties in $\gamma.\mathsf{users}$ are honest or if instructed by $\mathcal{S}$, execute steps (1)–(3) of the "All agreed" case for channels without validity with $\tau_3 := \tau_0 + T_1$.

Force close: Else in round τ_3 execute $\texttt{V–ForceClose}(\gamma.\mathsf{id})$.

Punishment (executed at the end of every round)

For every id, where $\gamma := \Gamma(\mathsf{id})$ is a virtual channel, set $(\mathsf{id}_A, \mathsf{id}_B) := \gamma.\mathsf{subchan}$. If this is the first round when $\Gamma(\mathsf{id}_A) = (\bot, \mathsf{tx}_A)$ or $\Gamma(\mathsf{id}_B) = (\bot, \mathsf{tx}_B)$, i.e., one of the subchannels was just closed, then let $\mathcal{S}$ set $t_1 \leq T'$, where $T' := \tau_0 + T + 5\Delta$ if $\gamma.\mathsf{val} = \bot$ and $T' := \gamma.\mathsf{val} + 3\Delta$ if $\gamma.\mathsf{val} \neq \bot$, and distinguish the following cases:

Offloaded: Latest in round t_1 the ledger $\mathcal{L}$ contains both

- a transaction tx_1 rooted at $\{\mathsf{tx}_A, \mathsf{tx}_B\}$ with an output $(\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_I)$. In this case $(\texttt{OFFLOADED}, \mathsf{id}) \xhookrightarrow{\tau_1} I$, where τ_1 is the round tx_1 appeared on $\mathcal{L}$.

- a transaction tx_2 with an output of value $\gamma.\mathsf{cash}$ and rooted at $\{\mathsf{tx}_A, \mathsf{tx}_B\}$, if $\gamma.\mathsf{val} = \bot$, and rooted at $\{\mathsf{tx}_A\}$, if $\gamma.\mathsf{val} \neq \bot$. Let τ_2 be the round when tx_2 appeared on $\mathcal{L}$. Then output $(\texttt{OFFLOADED}, \mathsf{id}) \overset{\tau_2}{\hookrightarrow} \gamma.\mathsf{users}$, set $\gamma' = \gamma$, $\gamma'.\mathsf{Ingrid} = \bot$, $\gamma'.\mathsf{subchan} = \bot$, $\gamma.\mathsf{val} = \bot$ and define $\Gamma(\mathsf{id}) := (\gamma', \mathsf{tx}_2)$.

Punished: Else for every honest party $P \in \gamma.\mathsf{users}$, check the following: the ledger $\mathcal{L}$ contains in round $\tau_1 \leq t_1$ a transaction tx rooted at either tx_A or tx_B with $(\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_P)$ as output. In that case, output $(\texttt{PUNISHED}, \mathsf{id}) \overset{\tau_1}{\hookrightarrow} P$. Set $\Gamma(\mathsf{id}) = \bot$ in the first round when PUNISHED was sent to all honest parties.

Error: If the above case is not true, then $(\texttt{ERROR}) \overset{t_1}{\hookrightarrow} \gamma.\mathsf{users}$.

V–ForceClose(id): Let τ_0 be the current round and $\gamma := \Gamma(\mathsf{id})$. Execute subprocedure Offload(id). Let $T' := \tau_0 + 2T_p + 8\Delta$ if $\gamma.\mathsf{val} = \bot$ and $T' := \gamma.\mathsf{val} + 3\Delta$ if $\gamma.\mathsf{val} \neq \bot$. If in round $\tau_1 \leq T'$ it holds that $\Gamma(\mathsf{id}) = (\gamma, \mathsf{tx})$, execute subprocedure L–ForceClose(id).

Subprocedure Offload(id): Let τ_0 be the current round, $\gamma := \Gamma(\mathsf{id})$, $(\mathsf{id}_\alpha, \mathsf{id}_\beta) := \gamma.\mathsf{subchan}$, $(\alpha, \mathsf{tx}_A) := \Gamma(\mathsf{id}_\alpha)$ and $(\beta, \mathsf{tx}_B) := \Gamma(\mathsf{id}_\beta)$. If within Δ rounds, neither tx_A nor tx_B is spent, then output $(\texttt{ERROR}) \overset{\tau_0+\Delta}{\hookrightarrow} \gamma.\mathsf{users}$.

Subprocedure UpdateState(id, θ): Let $(\alpha, \mathsf{tx}) := \Gamma(\mathsf{id})$. Set $\alpha.\mathsf{st} := \theta$ and update $\Gamma(\mathsf{id}) := (\alpha, \mathsf{tx})$.

E.4.2 Virtual Channels With Validity

We now briefly present our virtual channel protocol with validity. We focus mainly on the creation of the virtual channel as this illustrates the main structural differences to our construction without validity. For the full formal protocol description, we refer the reader to Appendix E.4.3.

Create. Unlike the without validity case, the structure of the construction with validity is not symmetric (see Figure E.2). The output of the ledger channel between A and I is used as the input for the funding transaction of the virtual channel tx^{f}, whereas the output of the channel between B and I is used for the so-called refund transaction $\mathsf{tx}_{\mathsf{refund}}$.

A can create tx^{f} on her own from the last state of her ledger channel with I. As a second step, A and B can already create the transactions required for the virtual channel γ. Additionally, I and B create the refund transaction which returns I's collateral if the virtual channel is offloaded. Finally, the created transactions are signed in reverse order. In particular, B signs $\mathsf{tx}_{\mathsf{refund}}$ so that I is ensured that she can publish it and receive her collateral and fees. Then, I signs tx^{f} and provides the signature to A, effectively authorizing her to publish tx^{f}, thereby allowing A to offload the virtual channel.

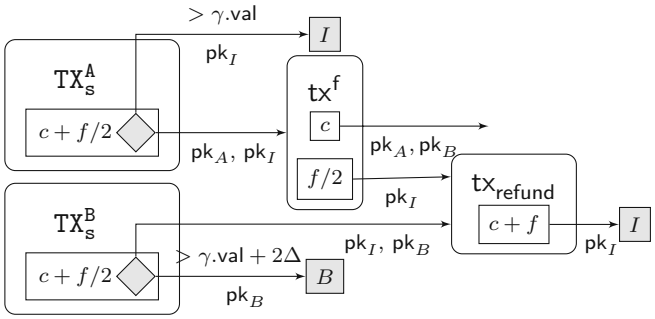

Figure E.2: Funding of a virtual channel γ with validity $\gamma.\mathsf{val}$.

Offload. In our virtual channel with validity, only A can offload the virtual channel γ by publishing the commit and split transaction of her ledger channel with I. Although I and B are not able to offload the virtual channel, they have the guarantee that after round $\gamma.\mathsf{val}$ either the channel is offloaded or closed or they can punish A and get reimbursed.

Punish. Recall that after a successful offload, the punishment mechanisms of generalized channels apply. We now discuss other malicious behaviors specific to this construction. In this protocol, only A can post the funding transaction of the virtual channel. If the virtual channel is not closed or offloaded by $\gamma.\mathsf{val}$, A is punished. A loses her coins to I and I loses her coins to B. Therefore, though B cannot offload the channel, he will get reimbursed from his ledger channel with I and I will get reimbursed regardless of whether the virtual channel is offloaded or not. At the time val, if the virtual channel is not honestly closed or the funding is not published, I submits the punishment transaction to reimburse her collateral. Therefore, at time $\mathsf{val} + \Delta$, either the punishment or the funding transaction is posted. If the virtual channel is offloaded, I can publish the refund transaction within Δ to get her coins back.

We mention here only for our virtual channel construction with validity. We refer the reader to Appendix E.5.1 for the full proof.

Theorem 18. *Let Σ be a signature scheme that is strongly unforgeable against chosen message attacks. Then for any ledger delay $\Delta \in \mathbb{N}$, the virtual channel protocol with validity as described in Appendix E.4.2 working in $\mathcal{F}_{preL}(3,1)$-hybrid, UC-realizes the ideal functionality $\mathcal{F}_V(3)$.*

E.4.3 Formal Virtual Channel Protocol

We now formally describe the protocol $\Pi_V(T_p)$ that was discussed on a high level in Section 6.3.4 and Appendix E.4.2. Since our goal is to prove that $\Pi_V(T_p)$ UC-realizes $\mathcal{F}_V(T_p)$, we need to discuss about how parties deal with instructions about ledger channels as well as virtual channels.

Ledger Channels. As a first step, we discuss how parties deal with messages about ledger channels or prepared ledger channels. On a high level, parties simply forward

these instructions to the hybrid ideal functionality $\mathcal{F}_{preL}(T_p, 1)$. If the functionality sends a reply, parties forward this reply to the environment. In addition to the message forwarding, parties store information about the ledger channels in a channel space Γ_L. More precisely, once a ledger channel is created or pre-created, parties add this channel to Γ_L. Once an existing ledger channel is updated or pre-updated, the party updates the latest state of the channel stored in Γ_L.

There is one technicality that we need to take care of. There are two different situations in which a party of a virtual channel protocol instructs the hybrid ideal functionality $\mathcal{F}_{preL}(T_p, 1)$ to pre-cerate (resp. pre-update) a channel γ:

1. Party receives a pre-create, resp. pre-update, instruction from the environment. As discussed above, in this case, the party acts as a dummy party and forwards the message to $\mathcal{F}_{preL}(T_p, 1)$.

2. Party is creating, resp. updating, a virtual channel and hence is sending pre-create, resp. pre-update, messages to $\mathcal{F}_{preL}(T_p, 1)$.

Let us stress that while channels pre-created via option (1) exist in both the real and ideal world, channels pre-created via option (2) exist only in the real world. This is because the pre-creation of these channels was not initiated by the environment but by the parties of the virtual channel protocol. Hence, we need to make sure that the environment cannot "accidentally" update a channel pre-created via (2) since this would help the environment distinguish between the real and ideal world.

To this end, party in the case (1) modifies the identifier of the channel by adding a prefix "ledger". More precisely, if the environment makes a request about a channel with identifier id it forwards the instruction to the hybrid functionality but replaces id with ledger$\|\mathsf{id}$. Analogously, if the hybrid functionality replies to this message, the party removes the prefix. This ensures that the environment cannot directly make any change on the ledger channels pre-created via option (2).

Virtual Channels. We now present the formal pseudocode for our virtual channel protocols $\Pi_V(T_p)$. As for ledger channels, the description excludes several checks that parties have to make. We formalize all these checks in the form of a functionality wrapper in Appendix E.5.

Create.

The creation of a virtual channel was described on a high level in Section 6.3.4. The main idea is to update the two subchannels of the virtual channel and pre-create a new ledger channel corresponding to the virtual channel. Importantly, the update of the subchannel needs to be synchronized in order to ensure that either both updates complete (in which case the virtual channel is created) or both updates are rejected (in which case the virtual channel creation fails).

Since a large part of the creation process is the same for channels with and without validity, our formal description is modularized.

Create a virtual channels - modular

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p,\ 1)$, $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$, $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{users}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$.

Party $P \in \{A, B\}$

Upon receiving $(\texttt{CREATE}, \gamma) \xleftarrow{t_0^P} \mathcal{E}$ proceed as follows:

1. Let $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(P)$ and compute
$$\theta_P := \texttt{GenVChannelOutput}(\gamma, P).$$
2. Send $(\texttt{UPDATE}, \mathsf{id}_\alpha, \theta_P, t_{\mathsf{stp}}) \xhookrightarrow{t_0^P} \mathcal{F}_{preL}$.
3. Upon receiving $(\texttt{SETUP}, \mathsf{id}_\alpha, \mathsf{tid}_P) \xleftarrow{t_1^P \le t_0^P + T} \mathcal{F}_{preL}$, engage in the subprotocol `SetupVChannel` with input (γ, tid_P).

Party I

Upon receiving $(\texttt{CREATE}, \gamma) \xleftarrow{t_0^I} \mathcal{E}$ proceed as follows:

1. Set $\mathsf{id}_\alpha = \gamma.\mathsf{subchan}(A), \mathsf{id}_\beta = \gamma.\mathsf{subchan}(B)$ and generate
$$\theta_A := \texttt{GenVChannelOutput}(\gamma, A)$$
$$\theta_B := \texttt{GenVChannelOutput}(\gamma, B)$$
2. If in round $t_1^I \le t_0^I + T$ you have received both $(\texttt{UPDATE\_REQ}, \mathsf{id}_\alpha, \theta_A, t_{\mathsf{stp}}, \mathsf{tid}_A) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{UPDATE\_REQ}, \mathsf{id}_\beta, \theta_B, t_{\mathsf{stp}}, \mathsf{tid}_B) \hookleftarrow \mathcal{F}_{preL}$, then engage in the subprotocol `SetupVChannel` with inputs $(\gamma, \mathsf{tid}_A, \mathsf{tid}_B)$. Else stop.

Party $P \in \{A, B\}$

Wait until $t_2^P := t_1^P + t_{\mathsf{stp}}$. If the subprotocol completed successfully, then send $(\texttt{SETUP\_OK}, \mathsf{id}_\alpha) \xhookrightarrow{t_2^P} \mathcal{F}_{preL}$. Else stop.

Party I

If in round $t_2^I \leq t_1^I + t_{\mathsf{stp}} + T$ you receive both $(\texttt{SETUP–OK}, \mathsf{id}_\alpha)) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{SETUP–OK}, \mathsf{id}_\beta)) \hookleftarrow \mathcal{F}_{preL}$, send $(\texttt{UPDATE–OK}, \mathsf{id}_\alpha) \overset{t_2}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\texttt{UPDATE–OK}, \mathsf{id}_\beta) \overset{t_2}{\hookrightarrow} \mathcal{F}_{preL}$. Otherwise stop.

Party $P \in \{A, B\}$

1. If you receive $(\texttt{UPDATE–OK}, \mathsf{id}_\alpha) \overset{t_2^P \leq t_1^P + 2T}{\longleftarrow} \mathcal{F}_{preL}$, reply with $(\texttt{REVOKE}, \mathsf{id}_\alpha) \overset{t_2^P + T}{\longrightarrow} \mathcal{F}_{preL}$. Otherwise stop.

Party I

If in round $t_3^I \leq t_2^I + 4T$ you have received both $(\texttt{REVOKE–REQ}, \mathsf{id}_\alpha) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{REVOKE–REQ}, \mathsf{id}_\beta) \hookleftarrow \mathcal{F}_{preL}$, reply $(\texttt{REVOKE}, \mathsf{id}_\alpha) \overset{t_3^I}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\texttt{REVOKE}, \mathsf{id}_\beta) \overset{t_3^I}{\hookrightarrow} \mathcal{F}_{preL}$ and update $\Gamma^I(\gamma.\mathsf{id})$ from $(\bot, x)$ to (γ, x). Otherwise stop.

Party $P \in \{A, B\}$

Upon receiving $(\texttt{UPDATED}, \mathsf{id}_\alpha) \overset{t_3^P \leq t_2^P + 3T}{\longleftarrow} \mathcal{F}_{preL}$, mark γ as created, i.e. update $\Gamma^P(\gamma.\mathsf{id})$ from $(\bot, x)$ to (γ, x), and output $(\texttt{CREATED}, \gamma.\mathsf{id}) \overset{t_3^P}{\hookrightarrow} \mathcal{E}$.

Function $\texttt{GenVChannelOutput}(\gamma, P)$

Return θ, where $\theta.\mathsf{cash} = \gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2$ and $\theta.\varphi$ is defined as follows

$$\theta.\varphi = \begin{cases} \texttt{Multi–Sig}_{\gamma.\mathsf{users}} \vee (\texttt{One–Sig}_P \wedge \texttt{CheckRelative}_{(T_p + 4\Delta)}), \\ \qquad\qquad \text{if } \gamma.\mathsf{val} = \bot \\ \texttt{Multi–Sig}_{A,I} \vee (\texttt{One–Sig}_I \wedge \texttt{CheckLockTime}_{\gamma.\mathsf{val}}), \\ \qquad\qquad \text{if } \gamma.\mathsf{val} \neq \bot \wedge P = A \\ \texttt{Multi–Sig}_{B,I} \vee (\texttt{One–Sig}_B \wedge \texttt{CheckLockTime}_{\gamma.\mathsf{val}+2\Delta}), \\ \qquad\qquad \text{if } \gamma.\mathsf{val} \neq \bot \wedge P = B \end{cases}$$

Subprotocol $\texttt{SetupVChannel}$

Let t_0 be the current round.

Channels without validity

Party $P \in \{A, B\}$ on input (γ, tid_P)

1. Create the body of the funding transactions:

$$\begin{aligned}\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{input} :=& (\mathsf{tid}_P, \mathsf{tid}_Q)\\ \mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{output} :=& ((\gamma.\mathsf{cash}, \texttt{Multi–Sig}_{\{\gamma.\mathsf{users}\}}),\\ & (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_{\mathsf{pk}_I}))\end{aligned}$$

2. Send $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, 1, t_{ofl}) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$, where $t_{ofl} = 2T_p + 8\Delta$.

3. If $(\texttt{PRE–CREATED}, \gamma.\mathsf{id}) \overset{t_1 \leq t_0+T}{\hookleftarrow} \mathcal{F}_{preL}$, then sign the funding transaction, i.e. $s_{\mathsf{f}}^P \leftarrow \mathsf{Sign}_{\mathsf{sk}_P}([\mathsf{tx}_{\mathsf{f}}^{\gamma}])$ and send $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^P, [\mathsf{tx}_{\mathsf{f}}^{\gamma}]) \overset{t_1}{\hookrightarrow} I$. Else stop.

Party I on input $(\gamma, \mathsf{tid}_A, \mathsf{tid}_B)$

4. If you receive $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^A, [\mathsf{tx}_{\mathsf{f}}^{\gamma}]) \overset{t_2 \leq t_0+T+1}{\hookleftarrow} A$ and $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^B, [\mathsf{tx}_{\mathsf{f}}^{\gamma}]) \overset{t_2}{\hookleftarrow} B$, verify the funding transaction and signatures of A and B, i.e. check:

$$\begin{aligned}\mathsf{Vrfy}_{\mathsf{pk}_A}([\mathsf{tx}_{\mathsf{f}}^{\gamma}]; s_{\mathsf{f}}^A) &= 1\\ \mathsf{Vrfy}_{\mathsf{pk}_B}([\mathsf{tx}_{\mathsf{f}}^{\gamma}], s_{\mathsf{f}}^B) &= 1\\ (\mathsf{tid}_A, \mathsf{tid}_B) &= \mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{input}\\ (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_{\mathsf{pk}_I}) &\in \mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{output}.\end{aligned}$$

5. If all checks pass, sign the funding transaction, i.e. compute

$$\begin{aligned}s_{\mathsf{f}}^I &:= \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}_{\mathsf{f}}^{\gamma}]),\\ \mathsf{tx}_{\mathsf{f}}^{\gamma} &:= \{([\mathsf{tx}_{\mathsf{f}}^{\gamma}], s_{\mathsf{f}}^A, s_{\mathsf{f}}^B, s_{\mathsf{f}}^I)\}.\end{aligned}$$

Store $\Gamma^I(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_{\mathsf{f}}^{\gamma})$. Then send $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^B, s_{\mathsf{f}}^I) \overset{t_2}{\hookrightarrow} A$ and $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^A, s_{\mathsf{f}}^I) \overset{t_2}{\hookrightarrow} B$, and consider procedure successfully completed. Else stop.

Party $P \in \{A, B\}$

6. Upon receiving $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^Q, s_{\mathsf{f}}^I) \overset{t_1+1}{\hookleftarrow} I$, verify all signatures, i.e. check:

$$\begin{aligned}\mathsf{Vrfy}_{\mathsf{pk}_Q}([\mathsf{tx}_{\mathsf{f}}^{\gamma}]; s_{\mathsf{f}}^Q) &= 1\\ \mathsf{Vrfy}_{\mathsf{pk}_I}([\mathsf{tx}_{\mathsf{f}}^{\gamma}], s_{\mathsf{f}}^I) &= 1.\end{aligned}$$

If all checks pass define $\mathsf{tx}_{\mathsf{f}}^{\gamma} := \{([\mathsf{tx}_{\mathsf{f}}^{\gamma}], s_{\mathsf{f}}^P, s_{\mathsf{f}}^Q, s_{\mathsf{f}}^I)\}$ and set $\Gamma^P(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_{\mathsf{f}}^{\gamma}, \mathsf{tid}_P)$ and consider procedure successfully completed. Else stop.

Channels with validity

Party A on input (γ, tid_A)

1. Send $(\text{createInfo}, \gamma.\mathsf{id}, \mathsf{tid}_A) \overset{t_0}{\hookrightarrow} B$
2. In round $t_1 := t_0 + 1$, create the body of the funding transaction:

$$\begin{aligned}\mathsf{tx}_\mathsf{f}^\gamma.\mathsf{input} :=& (\mathsf{tid}_A)\\ \mathsf{tx}_\mathsf{f}^\gamma.\mathsf{output} :=& ((\gamma.\mathsf{cash}, \texttt{Multi–Sig}_{\{\gamma.\mathsf{users}\}}),\\ & (\gamma.\mathsf{fee}/2, \texttt{One–Sig}_{\mathsf{pk}_I}))\end{aligned}$$

3. Send $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^\mathsf{f}, 1, t_{ofl}) \overset{t_1}{\hookrightarrow} \mathcal{F}_{preL}$, for $t_{ofl} = \gamma.\mathsf{val} + 3\Delta$.
4. If $(\texttt{PRE–CREATED}, \gamma.\mathsf{id}) \overset{t_2 \le t_1 + T}{\hookleftarrow} \mathcal{F}_{preL}$, then goto step (10). Else stop.

Party B on input (γ, tid_B)

5. If $(\text{createInfo}, \gamma.\mathsf{id}, \mathsf{tid}_A) \overset{t_1 := t_0 + 1}{\hookleftarrow} A$, then create the body of the funding and refund transactions:

$$\begin{aligned}\mathsf{tx}_\mathsf{f}^\gamma.\mathsf{input} :=& (\mathsf{tid}_A)\\ \mathsf{tx}_\mathsf{f}^\gamma.\mathsf{output} :=& ((\gamma.\mathsf{cash}, \texttt{Multi–Sig}_{\{\gamma.\mathsf{users}\}}),\\ & (\gamma.\mathsf{fee}/2, \texttt{One–Sig}_{\mathsf{pk}_I}))\\ \mathsf{tx}_{\mathsf{refund}}^\gamma.\mathsf{input} :=& (\mathsf{tx}_\mathsf{f}^\gamma.\mathsf{txid}||2, \mathsf{tid}_B)\\ \mathsf{tx}_{\mathsf{refund}}^\gamma.\mathsf{output} :=& (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_{\mathsf{pk}_I}).\end{aligned}$$

   Else stop.

6. Send $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^\mathsf{f}, 1, t_{ofl}) \overset{t_1}{\hookrightarrow} \mathcal{F}_{preL}$, for $t_{ofl} = \gamma.\mathsf{val} + 3\Delta$.
7. If $(\texttt{PRE–CREATED}, \gamma.\mathsf{id}) \overset{t_2 \le t_1 + T}{\hookleftarrow} \mathcal{F}_{preL}$, then compute a signature on the refund transaction, i.e., $s^B_{\mathtt{Ref}} \leftarrow \mathsf{Sign}_{\mathsf{sk}_B}([\mathsf{tx}^\gamma_{\mathsf{refund}}])$ and define $\Gamma^B(\gamma.\mathsf{id}) := (\bot, [\mathsf{tx}^\gamma_\mathsf{f}], \mathsf{tid}_B)$. Then, send $(\text{createFund}, \gamma.\mathsf{id}, s^B_{\mathtt{Ref}}, [\mathsf{tx}^\gamma_{\mathsf{refund}}], [\mathsf{tx}^\gamma_\mathsf{f}]) \overset{t_2}{\hookrightarrow} I$ and consider procedure successfully completed. Else stop.

Party I on input $(\gamma, \mathsf{tid}_A, \mathsf{tid}_B)$

8. If $(\text{createFund}, \gamma.\mathsf{id}, s^B_{\mathtt{Ref}}, [\mathsf{tx}^\gamma_{\mathsf{refund}}], [\mathsf{tx}^\gamma_\mathsf{f}]) \overset{t_3 \le t_0 + T + 2}{\hookleftarrow} B$, verify the fund and refund transactions and signature of B, i.e. check:

$$\begin{aligned}&\mathsf{Vrfy}_{\mathsf{sk}_B}([\mathsf{tx}^\gamma_{\mathsf{refund}}]; s^B_{\mathtt{Ref}}) = 1.\\ &[\mathsf{tx}^\gamma_{\mathsf{refund}}].\mathsf{input} = (\mathsf{tx}^\gamma_\mathsf{f}.\mathsf{txid}||2, \mathsf{tid}_B),\\ &[\mathsf{tx}^\gamma_{\mathsf{refund}}].\mathsf{output} = (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_{\mathsf{pk}_I}),\\ &[\mathsf{tx}^\gamma_\mathsf{f}].\mathsf{output}[2] = (\gamma.\mathsf{fee}/2, \texttt{One–Sig}_{\mathsf{pk}_I})\end{aligned}$$

If all checks pass, then sign the fund and refund transactions, i.e. compute

$$s^I_{\mathtt{Ref}} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}^\gamma_{\mathsf{refund}}]),\ s^I_{\mathtt{f}} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}^\gamma_{\mathtt{f}}]),$$
$$\mathsf{tx}^\gamma_{\mathsf{refund}} := \{([\mathsf{tx}^\gamma_{\mathsf{refund}}], s^I_{\mathtt{Ref}}, s^B_{\mathtt{Ref}})\}.$$

Else stop.

9. Store $\Gamma^I(\gamma.\mathsf{id}) := (\bot, [\mathsf{tx}^\gamma_{\mathtt{f}}], \mathsf{tx}^\gamma_{\mathsf{refund}}, \mathsf{tid}_A, \mathsf{tid}_B)$, send the message $(\mathsf{createFund}, \gamma.\mathsf{id}, s^I_{\mathtt{f}}) \overset{t_3}{\hookrightarrow} A$, and consider procedure successfully completed.

Party A

10. If you receive $(\mathsf{createFund}, \gamma.\mathsf{id}, s^I_{\mathtt{f}}) \overset{t_2+2}{\longleftarrow} I$, verify the signature, i.e. check $\mathsf{Vrfy}_{\mathsf{pk}_I}([\mathsf{tx}^\gamma_{\mathtt{f}}]; s^I_{\mathtt{f}}) = 1$. If the check passes, compute a signature on the fund transaction:

$$s^A_{\mathtt{f}} := \mathsf{Sign}_{\mathsf{sk}_A}([\mathsf{tx}^\gamma_{\mathtt{f}}]),$$
$$\mathsf{tx}^{\gamma,\mathsf{A}}_{\mathtt{f}} := \{([\mathsf{tx}^\gamma_{\mathtt{f}}], s^I_{\mathtt{f}}, s^A_{\mathtt{f}})\}.$$

and set $\Gamma^A(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}^{\gamma,\mathsf{A}}_{\mathtt{f}}, \mathsf{tid}_A)$. Then consider procedure successfully completed. Else stop.

Update. As discussed in Section 6.3.4, in order to update a virtual channel, parties update the corresponding prepared channel. This is done in a black-box way via the hybrid functionality $\mathcal{F}_{preL}$. Hence, parties act as dummy parties as forward update instructions (modified by adding PRE–) to the hybrid functionality $\mathcal{F}_{preL}$ and forward the replies of the functionality (modified by removing PRE–) to the environment. In case the update fails, parties offload the channel which allows to resolve disputes on-chain.

Update

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p, 1)$.

Initiating party P:

1. Upon $(\mathtt{UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{t_0}{\longleftarrow} \mathcal{E}$, $(\mathtt{PRE\text{–}UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
2. If $(\mathtt{PRE\text{–}SETUP}, \mathsf{id}, \mathsf{tid}_P) \overset{t_1 \le t_0+T}{\longleftarrow} \mathcal{F}_{preL}$, $(\mathtt{SETUP}, \mathsf{id}, \mathsf{tid}_P) \overset{t_1}{\hookrightarrow} \mathcal{E}$. Else stop.
3. If $(\mathtt{SETUP\text{–}OK}, \mathsf{id}) \overset{t_2 \le t_1+t_{\mathsf{stp}}}{\longleftarrow} \mathcal{E}$, $(\mathtt{PRE\text{–}SETUP\text{–}OK}, \mathsf{id}) \overset{t_2}{\hookrightarrow} \mathcal{E}$. Else stop.
4. Distinguish the following three cases:
   - If $(\mathtt{PRE\text{–}UPDATE\text{–}OK}, \mathsf{id}) \overset{t_3 \le t_2+T}{\longleftarrow} \mathcal{F}_{preL}$, $(\mathtt{UPDATE\text{–}OK}, \mathsf{id}) \overset{t_3}{\hookrightarrow} \mathcal{E}$.
   - If $(\mathtt{PRE\text{–}UPDATE\text{–}REJECT}, \mathsf{id}) \overset{t_3 \le t_2+T}{\longleftarrow} \mathcal{F}_{preL}$, then stop.
   - Else execute the procedure $\mathtt{Offload}^P(\mathsf{id})$ and stop.
5. If $(\mathtt{REVOKE}, \mathsf{id}) \overset{t_3}{\longleftarrow} \mathcal{E}$, $(\mathtt{PRE\text{–}REVOKE}, \mathsf{id}) \overset{t_3}{\hookrightarrow} \mathcal{F}_{preL}$. Else execute $\mathtt{Offload}^P(\mathsf{id})$ and stop.

6. If $(\texttt{PRE–UPDATED}, \mathsf{id}) \xleftarrow{t_4 \le t_3 + T} \mathcal{F}_{preL}$, update the channel space, i.e., let $\gamma := \Gamma^P(\mathsf{id})$, set $\gamma.\mathsf{st} := \overrightarrow{\theta}$ and $\Gamma(\mathsf{id}) := \gamma$. Then $(\texttt{UPDATED}, \mathsf{id}) \xhookrightarrow{t_4} \mathcal{F}_{preL}$. Else execute $\texttt{Offload}^P(\mathsf{id})$ and stop.

Reacting party Q

1. Upon $(\texttt{PRE–UPDATE–REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xleftarrow{\tau_0} \mathcal{F}_{preL}$, $(\texttt{UPDATE–REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xhookrightarrow{\tau_0} \mathcal{E}$.
2. If $(\texttt{PRE–SETUP–OK}, \mathsf{id}) \xleftarrow{\tau_1 \le \tau_0 + t_{\mathsf{stp}} + T} \mathcal{F}_{preL}$, $(\texttt{SETUP–OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{E}$. Else stop.
3. If $(\texttt{UPDATE–OK}, \mathsf{id}) \xleftarrow{\tau_1} \mathcal{E}$, $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{F}_{preL}$. Else stop.
4. If $(\texttt{PRE–REVOKE–REQ}, \mathsf{id}) \xleftarrow{\tau_2 \le \tau_1 + T} \mathcal{F}_{preL}$, $(\texttt{REVOKE–REQ}, \mathsf{id}) \xhookrightarrow{\tau_2} \mathcal{E}$. Else execute $\texttt{Offload}^Q(\mathsf{id})$ and stop.
5. If $(\texttt{REVOKE}, \mathsf{id}) \xleftarrow{\tau_2} \mathcal{E}$, $(\texttt{PRE–REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_2} \mathcal{F}_{preL}$. Else execute $\texttt{Offload}^Q(\mathsf{id})$ and stop.
6. Upon $(\texttt{PRE–UPDATED}, \mathsf{id}) \xleftarrow{\tau_3 \le \tau_2 + T} \mathcal{F}_{preL}$, update the channel space, i.e., let $\gamma := \Gamma^Q(\mathsf{id})$, set $\gamma.\mathsf{st} := \overrightarrow{\theta}$ and $\Gamma(\mathsf{id}) := \gamma$. Then $(\texttt{UPDATED}, \mathsf{id}) \xhookrightarrow{\tau_3} \mathcal{E}$.

Offload. As a next step, we define the offloading process which transforms a virtual channel into a ledger channel. Let us stress that offloading can be triggered either by the environment via a message `OFFLOAD` or internally by parties when executing an update or close. To avoid code repetition, we define a procedure $\texttt{Offload}^P(\mathsf{id})$ and instruct parties upon receiving $(\texttt{OFFLOAD}, \mathsf{id}) \xleftarrow{t_0} \mathcal{E}$ to simply call $\texttt{Offload}^P(\mathsf{id})$.

Since channels with validity are constructed in a different way than channels without validity, the procedure is defined for the two cases separately.

Subprocedure $\texttt{Offload}^P(\mathsf{id})$

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p, 1)$, $A := \gamma.\mathsf{Alice}$ and $B := \gamma.\mathsf{Bob}$ and $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{users}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$. Let t_0 be the current round.

Channels without validity

$P \in \{A, B\}$

1. Extract γ and $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^P(\mathsf{id})$ and tid_P, tid_Q from $\mathsf{tx}_\mathsf{f}^\gamma$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(P)$ and send $(\texttt{CLOSE}, \mathsf{id}_\alpha) \xhookrightarrow{t_0} \mathcal{F}_{preL}$.
2. If you receive $(\texttt{CLOSED}, \mathsf{id}_\alpha) \xleftarrow{t_1 \le t_0 + T + 3\Delta} \mathcal{F}_{preL}$, then continue. Else set $\Gamma^P(\gamma.\mathsf{id}) = \bot$ and stop.
3. Let $T_2 := t_1 + T + 3\Delta$ and distinguish:
   - If in round $t_2 \le T_2$ a transaction with tid_Q appeared on $\mathcal{L}$, then $(\mathsf{post}, \mathsf{tx}_\mathsf{f}^\gamma) \xhookrightarrow{t_2} \mathcal{L}$.

- Else in round T_2 create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_P$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{pun}}])$. Then $(\mathrm{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{T_2}{\hookrightarrow} \mathcal{L}$.

4. Let $T_3 := t_2 + \Delta$ and distinguish the following two cases:
   - The transaction $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, then update $\Gamma_L^P(\mathsf{id}) := \Gamma^P(\mathsf{id})$ and set $m :=$ offloaded.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, then set $m :=$ punished.
5. Set $\Gamma^P(\mathsf{id}) = \bot$ and return m in round t_3.

Party I

1. Extract γ and $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^I(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_{\mathsf{f}}^{\gamma}$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and send the messages $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
2. If you receive both messages $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{t_1^A \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$ and $(\mathtt{CLOSED}, \mathsf{id}_\beta) \overset{t_1^B \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$, publish $(\mathrm{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma}) \overset{t_1}{\hookrightarrow} \mathcal{L}$, where $t_1 := \max\{t_1^A, t_1^B\}$. Otherwise set $\Gamma^I(\mathsf{id}) = \bot$ and stop.
3. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$, then $\Gamma^I(\mathsf{id}) = \bot$ and return "offloaded".

Channels with validity

Party A

1. Extract γ, tid_A and $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^A(\mathsf{id})$. Let $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$ and send $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
2. If you receive $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{t_1 \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$, then post $(\mathrm{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{A}}) \overset{t_2}{\hookrightarrow} \mathcal{L}$. Otherwise, set $\Gamma^A(\gamma.\mathsf{id}) = \bot$ and stop.
3. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$, then update $\Gamma_L^A(\mathsf{id}) := \Gamma^A(\mathsf{id})$, $\Gamma^A(\mathsf{id}) := \bot$ and return "offloaded".

Party B

1. Extract γ, tid_B and $[\mathsf{tx}_{\mathsf{f}}^{\gamma}]$ from $\Gamma^B(\mathsf{id})$. Let $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and send $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
2. If you receive $(\mathtt{CLOSED}, \mathsf{id}_\beta) \overset{t_1 \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$, then continue. Otherwise, set $\Gamma^B(\gamma.\mathsf{id}) = \bot$ and stop.
3. Create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_B$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_B})$ and set the value $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_B}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $t_2 := \max\{t_1, \gamma.\mathsf{val} + 2\Delta\}$ and send $(\mathrm{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{t_2}{\hookrightarrow} \mathcal{L}$.

4. Let $T_3 := t_2 + \Delta$ and distinguish the following two cases:
   - A transaction with identifier $\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, then define $\Gamma_L^B(\mathsf{id}) := \Gamma^B(\mathsf{id})$ and set $m :=$ offloaded.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, set $m :=$ punished.
5. Set $\Gamma^B(\mathsf{id}) := \bot$ and return m in round t_3.

Party I

1. Extract γ, tid_A, tid_B, $\mathsf{tx}_{\mathsf{refund}}^{\gamma}$ and $[\mathsf{tx}_{\mathsf{f}}^{\gamma}]$ from $\Gamma^I(\mathsf{id})$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and send $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
2. If you receive both messages $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{t_1^A \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$ and $(\mathtt{CLOSED}, \mathsf{id}_\beta) \overset{t_1^B \leq t_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_{preL}$, then continue. Otherwise, set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and stop.
3. Create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_A$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})$ and set the value $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $t_2 := \max\{t_1^A, \gamma.\mathsf{val}\}$ and send $(\mathsf{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{t_2}{\hookrightarrow} \mathcal{L}$.
4. Let $T_3 := t_2 + \Delta$ and distinguish the following two cases:
   - A transaction with identifier $\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $t_3' \leq T_3$, send $(\mathsf{post}, \mathsf{tx}_{\mathsf{refund}}^{\gamma}) \overset{t_4}{\hookrightarrow} \mathcal{L}$ where $t_4 := \max\{t_1^B, t_3'\}$. Once $\mathsf{tx}_{\mathsf{refund}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $t_5 \leq t_4 + \Delta$, then define $m :=$ offloaded and $\Gamma^I(\gamma.\mathsf{id}) = \bot$.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $t_3'' \leq T_3$, then define $m :=$ punished and $\Gamma^I(\gamma.\mathsf{id}) = \bot$.
5. Return m in round t_6 where $t_6 := \max\{t_5, t_3''\}$.

Close. In order to close a virtual channel, parties first try to adjust the balances in the subchannel according to the latest valid state of the virtual channel. This is done by updating the subchannels in a synchronous way, as was done during virtual channel creation. In case this process fails, parties close the channel forcefully. This means that parties first offload the channel and then immediately close the offloaded ledger channel.

Close a virtual channel

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p, 1)$, $A := \gamma.\mathsf{Alice}$ and $B := \gamma.\mathsf{Bob}$ and $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{users}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$.

Party $P \in \{A, B\}$

Upon receiving $(\mathtt{CLOSE}, \mathsf{id}) \overset{t_0^P}{\hookleftarrow} \mathcal{E}$ or in round $t_0^P := \gamma.\mathsf{val} - (4\Delta + 7T_p)$ if $\gamma.\mathsf{val} \neq \bot$, proceed as follows:

1. Extract γ, $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^P(\mathsf{id})$.
2. Parse $\gamma.\mathsf{st} = \Big((c_P, \texttt{One-Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One-Sig}_{\mathsf{pk}_Q})\Big)$.
3. Compute the new state of the channel $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(P)$ as

$$\overrightarrow{\theta}_P := \{(c_P, \texttt{One-Sig}_{\mathsf{pk}_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \texttt{One-Sig}_{\mathsf{pk}_I})\}$$

   Then, send $(\texttt{UPDATE}, \mathsf{id}_\alpha, \overrightarrow{\theta}_P, 0) \overset{t_0^P}{\hookrightarrow} \mathcal{F}_{preL}$.
4. Upon $(\texttt{SETUP}, \mathsf{id}_\alpha, \mathsf{tid}_P) \overset{t_1^P \leq t_0^P + T}{\hookleftarrow} \mathcal{F}_{preL}$, send $(\texttt{SETUP-OK}, \mathsf{id}_\alpha) \overset{t_1^P}{\hookrightarrow} \mathcal{F}_{preL}$.

Party I

Upon receiving $(\texttt{CLOSE}, \mathsf{id}) \overset{t_0^I}{\hookleftarrow} \mathcal{E}$ or in round $t_0^I := \gamma.\mathsf{val} - (4\Delta + 7T_p)$, proceed as follows:

1. Extract γ, $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^I(\mathsf{id})$.
2. Let $\mathsf{id}_\alpha = \gamma.\mathsf{subchan}(A), \mathsf{id}_\beta = \gamma.\mathsf{subchan}(B)$ and $c := \gamma.\mathsf{cash}$
3. If in round $t_1^I \leq t_0^I + T$ you received both $(\texttt{UPDATE-REQ}, \mathsf{id}_\alpha, \mathsf{tid}_A, \overrightarrow{\theta}_A, 0) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{UPDATE-REQ}, \mathsf{id}_\beta, \mathsf{tid}_B, \overrightarrow{\theta}_B, 0) \hookleftarrow \mathcal{F}_{preL}$ check that for some c_A, c_B s.t. $c_A + c_B = c$ it holds

$$\overrightarrow{\theta}_A = \{(c_A, \texttt{One-Sig}_{\mathsf{pk}_A}), (c_B + \gamma.\mathsf{fee}/2, \texttt{One-Sig}_{\mathsf{pk}_I})\}$$
$$\overrightarrow{\theta}_B = \{(c_B, \texttt{One-Sig}_{\mathsf{pk}_B}), (c_A + \gamma.\mathsf{fee}/2, \texttt{One-Sig}_{\mathsf{pk}_I})\}$$

   If not, then stop.
4. If in round $t_2^I \leq t_1^I + T$ you receive both $(\texttt{SETUP-OK}, \mathsf{id}_\alpha)) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{SETUP-OK}, \mathsf{id}_\beta)) \hookleftarrow \mathcal{F}_{preL}$, send $(\texttt{UPDATE-OK}, \mathsf{id}_\alpha) \overset{t_2^I}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\texttt{UPDATE-OK}, \mathsf{id}_\beta) \overset{t_2^I}{\hookrightarrow} \mathcal{F}_{preL}$. If not, then stop.

Party $P \in \{A, B\}$

If you receive $(\texttt{UPDATE-OK}, \mathsf{id}_\alpha) \overset{t_2^P \leq t_1^P + 2T}{\hookleftarrow} \mathcal{F}_{preL}$, reply with $(\texttt{REVOKE}, \mathsf{id}_\alpha) \overset{t_2^P}{\hookrightarrow} \mathcal{F}_{preL}$. Otherwise execute $\texttt{Offload}^P(\mathsf{id})$ and stop.

Party I

If in round $t_3^I \leq t_2^I + 2T$ you received both $(\texttt{REVOKE-REQ}, \mathsf{id}_\alpha) \hookleftarrow \mathcal{F}_{preL}$ and $(\texttt{REVOKE-REQ}, \mathsf{id}_\beta) \hookleftarrow \mathcal{F}_{preL}$, reply $(\texttt{REVOKE}, \mathsf{id}_\alpha) \overset{t_3^I}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\texttt{REVOKE}, \mathsf{id}_\beta) \overset{t_3^I}{\hookrightarrow} \mathcal{F}_{preL}$ and set $\Gamma^I(\mathsf{id}) := \bot$.

Party $P \in \{A, B\}$

If you receive $(\texttt{UPDATED}, \mathsf{id}_\alpha) \xleftarrow{t_3^P \le t_2^P + 2T} \mathcal{F}_{preL}$, set $\Gamma^P(\mathsf{id}) := \bot$. Then output $(\texttt{CLOSED}, \mathsf{id}) \xhookrightarrow{t_3^P} \mathcal{E}$ and stop. Else execute $\texttt{Offload}^P(\mathsf{id})$ and stop.

Punish. Finally, we formalize the actions taken by parties in every round. On a high level, in addition to triggering the hybrid ideal functionality to take the every-round actions for ledger channels (which include blockchain monitoring for outdated commit transactions), parties also need to make several checks for virtual channels. Namely, channel users who tried to offload the virtual channel by closing their subchannel) monitor whether the other subchannel was closed as well. If yes, then they can publish the funding transaction and complete the offload and otherwise apply the punishment mechanism.

Punish virtual channel

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p, 1)$, $A := \gamma.\mathsf{Alice}$ and $B := \gamma.\mathsf{Bob}$ and $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{users}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$.

Upon receiving $(\texttt{PUNISH}) \xleftarrow{\tau_0} \mathcal{E}$, do the following:

- Forward this message to the hybrid ideal functionality $(\texttt{PUNISH}) \xhookrightarrow{\tau_0} \mathcal{F}_{preL}$. If $(\texttt{PUNISHED}, \mathsf{id}) \xleftarrow{\tau_1} \mathcal{F}_{preL}$, then $(\texttt{PUNISHED}, \mathsf{id}) \xhookrightarrow{\tau_1} \mathcal{E}$.
- Execute both subprotocols `Punish` and `Punish–Validity`.

Punish

Party $P \in \{A, B\}$

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} = \bot$ can be extracted from $\Gamma^P(\mathsf{id})$ do the following:

1. Extract $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^P(\mathsf{id})$ and tid_P, tid_Q from $\mathsf{tx}_\mathsf{f}^\gamma$. Check if tid_P appeared on $\mathcal{L}$. If not, then stop.
2. Denote $T_2 := t_1 + T + 3\Delta$ and distinguish:
   - If in round $t_2 \le T_2$ the transaction with tid_Q appeared on $\mathcal{L}$, then $(\mathsf{post}, \mathsf{tx}_\mathsf{f}^\gamma) \xhookrightarrow{t_2} \mathcal{L}$.
   - Else in round T_2 create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as

$$\begin{aligned} \mathtt{TX}_{\mathsf{pun}}.\mathsf{input} &:= \mathsf{tid}_P \\ \mathtt{TX}_{\mathsf{pun}}.\mathsf{output} &:= (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P}) \\ \mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} &:= \mathsf{Sign}_{\mathsf{sk}_P}([\mathtt{TX}_{\mathsf{pun}}]), \end{aligned}$$

and $(\text{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{T_2}{\hookrightarrow} \mathcal{L}$.

3. Let $T_3 := t_2 + \Delta$ and distinguish the following two cases:
   - The transaction $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, then $\Gamma_L^P(\mathsf{id}) := \Gamma^P(\mathsf{id})$, $\Gamma^P(\mathsf{id}) = \bot$ and $m := \mathtt{OFFLOADED}$.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $t_3 \leq T_3$, then define $\Gamma^P(\gamma.\mathsf{id}) = \bot$ and set $m := \mathtt{PUNISHED}$.
4. Output $(m, \mathsf{id}) \overset{t_3}{\hookrightarrow} \mathcal{E}$.

Party I

For every $\mathsf{id} \in \{0,1\}^*$, such that γ with $\gamma.\mathsf{val} = \bot$ can be extracted from $\Gamma^I(\mathsf{id})$ do the following:

1. Extract $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^I(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_{\mathsf{f}}^{\gamma}$. Check if for some $P \in \{A, B\}$ a transaction with identifier tid_P appeared on $\mathcal{L}$. If not, then stop.
2. Denote $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(Q)$ and send $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$.
3. If you receive $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{t_1 \leq t_0 + T + 3\Delta}{\longleftarrow} \mathcal{F}_{preL}$ and tid_Q appeared on $\mathcal{L}$, $(\text{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma}) \overset{t_1}{\hookrightarrow} \mathcal{L}$. Otherwise set $\Gamma^I(\mathsf{id}) = \bot$ and stop.
4. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round t_2, such that $t_2 \leq t_1 + \Delta$, set $\Gamma^I(\mathsf{id}) = \bot$ and output $(\mathtt{OFFLOADED}, \mathsf{id}) \overset{t_2}{\hookrightarrow} \mathcal{E}$.

Punish–Validity

Party A

For every $\mathsf{id} \in \{0,1\}^*$, such that γ with $\gamma.\mathsf{val} \neq \bot$ can be extracted from $\Gamma^A(\mathsf{id})$ do the following:

1. Extract $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^A(\mathsf{id})$ and tid_A from $\mathsf{tx}_{\mathsf{f}}^{\gamma}$. If tid_A appeared on $\mathcal{L}$, then send $(\text{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma}) \overset{t_1}{\hookrightarrow} \mathcal{L}$. Else stop.
2. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $t_2 \leq t_1 + \Delta$, set $\Gamma_L^A(\mathsf{id}) := \Gamma^A(\mathsf{id})$, $\Gamma^A(\mathsf{id}) := \bot$ and output $(\mathtt{OFFLOADED}, \mathsf{id}) \overset{t_2}{\hookrightarrow} \mathcal{E}$.

Party B

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} = \bot$ can be extracted from $\Gamma^B(\mathsf{id})$ do the following:

1. Extract tid_B and $[\mathsf{tx}_\mathsf{f}^\gamma]$ from $\Gamma^B(\mathsf{id})$. Check if tid_B or $[\mathsf{tx}_\mathsf{f}^\gamma].\mathsf{txid}$ appeared on $\mathcal{L}$. If not, then stop.
2. If a transaction $\mathsf{tx}_\mathsf{f}^\gamma$ appeared on $\mathcal{L}$, update set $\Gamma_L^B(\mathsf{id}) := \Gamma^B(\mathsf{id})$ and $\Gamma^B(\mathsf{id}) := \bot$. Then output $(\texttt{OFFLOADED}, \mathsf{id}) \overset{t_1}{\hookrightarrow} \mathcal{E}$ and stop.
3. If tid_B appeared on $\mathcal{L}$, create the punishment transaction $\mathtt{TX}_\mathsf{pun}$ as

$$\begin{aligned}\mathtt{TX}_\mathsf{pun}.\mathsf{input} &:= \mathsf{tid}_B\\ \mathtt{TX}_\mathsf{pun}.\mathsf{output} &:= (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_B})\\ \mathtt{TX}_\mathsf{pun}.\mathsf{Witness} &:= \mathsf{Sign}_{\mathsf{sk}_B}([\mathtt{TX}_\mathsf{pun}]).\end{aligned}$$

   Then wait until round $t_2 := \max\{t_1, \gamma.\mathsf{val} + 2\Delta\}$ and send $(\mathsf{post}, \mathtt{TX}_\mathsf{pun}) \overset{t_2}{\hookrightarrow} \mathcal{L}$.
4. If transaction $\mathtt{TX}_\mathsf{pun}$ was accepted by $\mathcal{L}$ in $t_3 \leq t_2 + \Delta$, then define $\Gamma^B(\gamma.\mathsf{id}) = \bot$ and output $(\texttt{PUNISHED}, \mathsf{id}) \overset{t_3}{\hookrightarrow} \mathcal{E}$.

Party I

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} = \bot$ can be extracted from $\Gamma^I(\mathsf{id})$ do the following:

1. Extract tid_A, tid_B, $\mathsf{tx}_\mathsf{refund}^\gamma$ and $[\mathsf{tx}_\mathsf{f}^\gamma]$ from $\Gamma^I(\mathsf{id})$. Check if tid_A or tid_B appeared on $\mathcal{L}$ or $t_1 = \gamma.\mathsf{val} - (3\Delta + T_p)$. If not, then stop.
2. Distinguish the following cases:
   - If $t_1 = \gamma.\mathsf{val} - (3\Delta + T_p)$, define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and send $(\texttt{CLOSE}, \mathsf{id}_\alpha) \overset{t_1}{\hookrightarrow} \mathcal{F}_{preL}$ and $(\texttt{CLOSE}, \mathsf{id}_\beta) \overset{t_1}{\hookrightarrow} \mathcal{F}_{preL}$.
   - If tid_B appeared on $\mathcal{L}$, send $(\texttt{CLOSE}, \mathsf{id}_\alpha) \overset{t_1}{\hookrightarrow} \mathcal{F}_{preL}$.
3. If a transaction with identifier tid_A appeared on $\mathcal{L}$ in round $t_2 \leq t_1 + T + 3\Delta$, create the punishment transaction $\mathtt{TX}_\mathsf{pun}$ as

$$\begin{aligned}\mathtt{TX}_\mathsf{pun}.\mathsf{input} &:= \mathsf{tid}_A\\ \mathtt{TX}_\mathsf{pun}.\mathsf{output} &:= (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})\\ \mathtt{TX}_\mathsf{pun}.\mathsf{Witness} &:= \mathsf{Sign}_{\mathsf{sk}_I}([\mathtt{TX}_\mathsf{pun}]).\end{aligned}$$

   Then wait until round $t_3 := \max\{t_2, \gamma.\mathsf{val}\}$ and send $(\mathsf{post}, \mathtt{TX}_\mathsf{pun}) \overset{t_3}{\hookrightarrow} \mathcal{L}$.
4. Distinguish the following two cases:
   - The transaction $\mathsf{tx}_\mathsf{f}^\gamma.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $t_4 \leq t_3 + \Delta$, send $(\mathsf{post}, \mathsf{tx}_\mathsf{refund}^\gamma) \overset{t_5}{\hookrightarrow} \mathcal{L}$ where $t_5 := \max\{\gamma.\mathsf{val} + \Delta, t_4\}$. Once $\mathsf{tx}_\mathsf{refund}^\gamma$ is accepted by $\mathcal{L}$ in round $t_6 \leq t_5 + \Delta$, set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and output $(\texttt{OFFLOADED}, \mathsf{id}) \overset{t_6}{\hookrightarrow} \mathcal{E}$ and stop.

- The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $t_4 \leq t_3 + \Delta$, then set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and output $(\mathtt{PUNISHED}, \mathsf{id}) \overset{t_4}{\hookrightarrow} \mathcal{E}$.

E.5 Wrappers for Missing Checks

In the previous sections, we provided simplified descriptions of the ideal functionalities $\mathcal{F}_L$, $\mathcal{F}_{preL}$ and $\mathcal{F}_V$ as well as of the protocols Π_{preL} and Π_V. The simplification stems from the fact that we excluded several natural checks in the ideal functionalities and protocols. In this section, we present wrappers that include these missing checks.

Wrapper for Ideal Functionalities

In order to simplify the exposition, the formal descriptions of the channel ideal functionalities $\mathcal{F}_L$, $\mathcal{F}_{preL}$, and $\mathcal{F}_V$ are simplified. Namely, they exclude several natural checks that one would expect an ideal functionality to make when it receives a message from a party. The purpose of the checks is to avoid the functionality from accepting malformed messages. To provide some intuition, we present several examples of such restrictions:

- A party sends a malformed message (e.g. missing or additional parameters)
- A party requests creation of a virtual channel but one of the two subchannels does not exist or does not have enough funds for virtual channel creation.
- Parties try to update the same channel twice in parallel.

We now list all checks formally in the wrapper below which can be seen as an extension to the wrapper provided by [AEE+21] for $\mathcal{F}_L$.

Functionality wrapper: $\mathcal{W}_{\text{checks}}(T_p)$

The wrapper is defined for $\mathcal{F} \in \{\mathcal{F}_V(T_p), \mathcal{F}_{preL}(T_p), \mathcal{F}_L(T_p)\}$. Below, we abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Create: Upon $(\mathtt{CREATE}, \gamma, \mathsf{tid}) \overset{\tau_0}{\hookleftarrow} P$, where $P \in \gamma.\mathsf{users}$, check if: $\Gamma(\gamma.\mathsf{id}) = \bot$, $\mathcal{F}.\Gamma_{pre}(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created or pre-created; γ is valid according to the definition given in Section 6.3.1; $\gamma.\mathsf{st} = \{(c_P, \mathtt{One\text{–}Sig}_{\mathsf{pk}_P}), (c_Q, \mathtt{One\text{–}Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$. Depending on the type of channel, make the following additional checks:

ledger channel: There exists $(t, \mathsf{id}, i, \theta) \in \mathcal{L}.\mathtt{UTXO}$ such that $\theta = (c_P, \mathtt{One\text{–}Sig}_P)$ for $(\mathsf{id}, i) := \mathsf{tid}$;[a]

virtual channel:

- If $P \in \gamma.\mathsf{users}$, then $\alpha := \mathcal{F}.\Gamma(\mathsf{id}_P) \neq \bot$ for $\mathsf{id}_P := \gamma.\mathsf{subchan}(P)$; $\alpha.\mathsf{users} = \{P, I\}$; there is no other virtual channel being created over α and α is currently not being updated; both P and I have enough funds in α.

- If $P = I$, then $\alpha := \mathcal{F}.\Gamma(\mathsf{id}_A) \neq \bot$ for $\mathsf{id}_A := \gamma.\mathsf{subchan}(A)$; $\beta := \mathcal{F}.\Gamma(\mathsf{id}_B) \neq \bot$ for $\mathsf{id}_B := \gamma.\mathsf{subchan}(B)$; $\alpha.\mathsf{users} = \{A, I\}$; $\beta.\mathsf{users} = \{B, I\}$; there is no other virtual channel being created over α or β; A and I have enough funds in α and B and I have enough funds in β.
- If $\gamma.\mathsf{val} \neq \bot$, then $\gamma.\mathsf{val} \geq \tau_0 + 4\Delta + 15T_p$.

If one of the above checks fails, drop the message. Else proceed as $\mathcal{F}$.

Pre-Create: Upon $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{\mathit{oft}}) \overset{\tau_0}{\hookleftarrow} P$, check if: $P \in \gamma.\mathsf{users}$, $\mathcal{F}.\Gamma_{\mathit{pre}}(\gamma.\mathsf{id}) = \bot$, $\mathcal{F}.\Gamma_{\mathit{pre}}(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created or pre-created; γ is valid according to the definition given in Section 6.3.1; $\gamma.\mathsf{st} = \{(c_P, \texttt{One–Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One–Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$ and tx^{f} is not a published transaction on $\mathcal{L}$. If one of the above checks fails, drop the message. Else proceed as $\mathcal{F}$.

(Pre)-Update: Upon $(m, \mathsf{id}, \theta, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, check if: $\gamma := \Gamma(\mathsf{id}) \neq \bot$ if $m = \texttt{UPDATE}$ and $\gamma := \Gamma_{\mathit{pre}}(\mathsf{id}) \neq \bot$ if $m = \texttt{PRE–UPDATE}$. In both cases additionally check: $P \in \gamma.\mathsf{users}$; there is no other update being performed on γ; let $\theta = (\theta_1, \ldots \theta_\ell) = ((c_1, \varphi_1), \ldots, (c_\ell, \varphi_\ell))$, then $\sum_{j \in [\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $((\texttt{PRE–})\texttt{SETUP–OK}, \mathsf{id}) \overset{\tau_2}{\hookleftarrow} P$ check if: you accepted a message $((\texttt{PRE–})\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, where $t_2 - t_0 \leq t_{\mathsf{stp}} + T_p$ and the message is a reply to the message $((\texttt{PRE–})\texttt{SETUP}, \mathsf{id}, \mathsf{tid})$ sent to P in round τ_1 such that $\tau_2 - \tau_1 \leq t_{\mathsf{stp}}$[b]. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $((\texttt{PRE–})\texttt{UPDATE–OK}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to the message $((\texttt{PRE–})\texttt{SETUP–OK}, \mathsf{id})$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.

Upon $((\texttt{PRE–})\texttt{REVOKE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if the message is a reply to either the message $((\texttt{PRE–})\texttt{UPDATE–OK}, \mathsf{id})$ sent to P in round τ_0 or the message $((\texttt{PRE–})\texttt{REVOKE–REQ}, \mathsf{id})$ sent to P in round τ_0. If not, drop the message. Else proceed as $\mathcal{F}$.

Offload: Upon receiving $(\texttt{OFFLOAD}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$ make the following checks: $\gamma := \Gamma(\mathsf{id}) \neq \bot$ is a virtual channel and $P \in \gamma.\mathsf{users}$. If one of the checks fails, then drop the message. Otherwise proceed as the functionality $\mathcal{F}$.

Close: Upon $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} P$, check if $\gamma := \Gamma(\mathsf{id}) \neq \bot$ and $P \in \gamma.\mathsf{users}$. If γ is a virtual channel, additionally check that $\gamma.\mathsf{val} = \bot$. If not, drop the message. Else proceed as $\mathcal{F}$.

All other messages are dropped.

[a] In case more channels are being created at the same time, then none of the other creation requests can use of the tid.

[b] See Appendix E.1 what we formally meant by "reply".

Wrapper for Protocols

Similar to the descriptions of our ideal functionality, the description of our channel protocols, the protocol Π_{preL} presented in Appendix E.3.4 and the protocol Π_V, exclude many natural checks that we would want an honest party to make. Let us give a few examples of requests which an honest party drops if received from the environment:

- The environment sends a malformed message to a party P (e.g. missing or additional

parameters);

- A party P receives an instruction to create a channel γ but $P \notin \gamma.\mathsf{users}$;
- A party P receives an instruction to create a virtual channel on top of a ledger channel that does not exist, does not belong to part P or is not sufficiently funded.
- Parties request to create a channel with validity whose validity time already expired (or is about to expire).

We define all these checks as a wrapper $\mathcal{W}_{\text{checksP}}$ that can be seen as an extension of the wrapper provided by [AEE+21] for their ledger channel protocol.

Protocol wrapper: $\mathcal{W}_{\text{checksP}}$

The wrapper is defined for $\Pi \in \{\Pi_V(T_p),\ \Pi_{preL}(T_p)\}$. Below, we abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$ and $I := \gamma.\mathsf{Ingrid}$.

Party P

Create: Upon $(\texttt{CREATE}, \gamma, \mathsf{tid}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ check if: $P \in \gamma.\mathsf{users}$; $\Gamma^P(\gamma.\mathsf{id}) = \bot$, $\Gamma^P_{pre}(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created or pre-created; γ is valid according to the definition given in Section 6.3.1; $\gamma.\mathsf{st} = \{(c_P, \texttt{One–Sig}_{\mathsf{pk}_P}), (c_Q, \texttt{One–Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$. Depending on the type of channel, make the following additional checks:

ledger channel: There exists $(t,\ \mathsf{id},\ i,\ \theta) \in \mathcal{L}.\texttt{UTXO}$ such that $\theta = (c_P,\ \texttt{One–Sig}_P)$ for $(\mathsf{id},\ i) := \mathsf{tid}$;[a]

virtual channel:

- If $P \in \gamma.\mathsf{users}$, then $\alpha := \Gamma^P(\mathsf{id}_P) \neq \bot$ for $\mathsf{id}_P := \gamma.\mathsf{subchan}(P)$; $\alpha.\mathsf{users} = \{P,\ I\}$; there is no other virtual channel being created over α and α is currently not being updated; both P and I have enough funds in α.
- If $P = I$, then $\alpha := \Gamma^P(\mathsf{id}_A) \neq \bot$ for $\mathsf{id}_A := \gamma.\mathsf{subchan}(A)$; $\beta := \Gamma^P(\mathsf{id}_B) \neq \bot$ for $\mathsf{id}_B := \gamma.\mathsf{subchan}(B)$; $\alpha.\mathsf{users} = \{A,\ I\}$; $\beta.\mathsf{users} = \{B,\ I\}$; there is no other virtual channel being created over α or β; A and I have enough funds in α and B and I have enough funds in β.
- If $\gamma.\mathsf{val} \neq \bot$, then $\gamma.\mathsf{val} \geq \tau_0 + 4\Delta + 15T_p$.

If one of the checks fails, drop the message. Else proceed as in Π.

Pre-Create: Upon $(\texttt{PRE–CREATE}, \gamma,\ \mathsf{tx}^{\mathsf{f}},\ i,\ t_{ofl}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if: $P \in \gamma.\mathsf{users}$, $\Gamma^P_{pre}(\gamma.\mathsf{id}) = \bot$, $\Gamma^P_{pre}(\gamma.\mathsf{id}) = \bot$ and there is no channel γ' with $\gamma.\mathsf{id} = \gamma'.\mathsf{id}$ being created or pre-created; γ is valid according to the definition given in Section 6.3.1; $\gamma.\mathsf{st} = \{(c_P,\ \texttt{One–Sig}_{\mathsf{pk}_P}),\ (c_Q,\ \texttt{One–Sig}_{\mathsf{pk}_Q})\}$ for $c_P, c_Q \in \mathbb{R}^{\geq 0}$ and tx^{f} is not a published transaction on $\mathcal{L}$. If one of the above checks fails, drop the message. Else proceed as in Π.

(Pre)-Update: Upon $(m,\ \mathsf{id},\ \theta,\ t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ check if: $\gamma := \Gamma^P(\mathsf{id}) \neq \bot$ if $m = \texttt{UPDATE}$ and $\gamma := \Gamma^P_{pre}(\mathsf{id}) \neq \bot$ if $m = \texttt{PRE–UPDATE}$. In both cases, check that there is no other update being preformed on γ; let $\theta = (\theta_1,\ \ldots \theta_\ell) = ((c_1,\ \varphi_1),\ \ldots,\ (c_\ell,\ \varphi_\ell))$, then $\sum_{j\in[\ell]} c_i = \gamma.\mathsf{cash}$ and $\varphi_j \in \mathcal{L}.\mathcal{V}$ for each $j \in [\ell]$. If on of the checks fails, drop the message. Else proceed as in Π.

Upon $((\texttt{PRE–})\texttt{SETUP–OK}, \mathsf{id}) \overset{\tau_2}{\hookleftarrow} \mathcal{E}$ check if: you accepted a message $((\texttt{PRE–})\texttt{UPDATE}, \mathsf{id}, \theta, t_{\mathsf{stp}})$ $\overset{\tau_0}{\hookleftarrow} \mathcal{E}$, where $t_2 - t_0 \leq t_{\mathsf{stp}} + T_p$ and the message is a reply to the message $((\texttt{PRE–})\texttt{SETUP}, \mathsf{id}, \mathsf{tid})$ you sent in round τ_1 such that $\tau_2 - \tau_1 \leq t_{\mathsf{stp}}$[b]. If not, drop the message. Else proceed as in Π.

Upon $((\texttt{PRE–})\texttt{UPDATE–OK}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if the message is a reply to the message $((\texttt{PRE–})\texttt{SETUP–OK}, \mathsf{id})$ you sent in round τ_0. If not, drop the message. Else proceed as in Π.

Upon $((\texttt{PRE–})\texttt{REVOKE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if the message is a reply to either $((\texttt{PRE–})\texttt{UPDATE–OK}, \mathsf{id})$ or $((\texttt{PRE–})\texttt{REVOKE–REQ}, \mathsf{id})$ you sent in round τ_0. If not, drop the message. Else proceed as in Π.

Offload: Upon receiving $(\texttt{OFFLOAD}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$ make the following checks: $\gamma := \Gamma(\mathsf{id}) \neq \bot$ is a virtual channel and $P \in \gamma.\mathsf{users}$. If one of the checks fails, then drop the message. Else proceed as in Π.

Close: Upon $(\texttt{CLOSE}, \mathsf{id}) \overset{\tau_0}{\hookleftarrow} \mathcal{E}$, check if $\gamma := \Gamma^P(\mathsf{id}) \neq \bot$, $P \in \gamma.\mathsf{users}$. If γ is a virtual channel, additionally check that $\gamma.\mathsf{val} = \bot$. If not, drop the message. Else proceed as in Π.

All other messages are dropped.

[a] In case more channels are being created at the same time, then none of the other creation requests can use of the tid.

[b] See Appendix E.1 what we formally meant by "reply".

E.5.1 Security Proofs

In this section, we provide proofs for Theorem 17, Theorem 6 and Theorem 18.

Proof of Theorem 17

In our proof of Theorem 17, we provide the code for a simulator, that simulates the protocol Π_{preL} in the ideal world, having access to the functionalities $\mathcal{L}$ and $\mathcal{F}_{preL}$. In UC proofs it is required to provide a simulation of the real protocol in the ideal world even without knowledge of the secret inputs of the honest protocol participants. The main challenge is that this transcript of the simulation has to be indistinguishable to the environment $\mathcal{E}$ from the transcript of the real protocol execution. Yet, in our protocols, parties do not receive secret inputs but are only instructed by the environment to take certain protocol actions, e.g. updating a channel. Hence the only challenge that arises during simulation is handling different behaviors of malicious parties. Due to this, we only provide the simulator code for the protocol without arguing about indistinguishability of simulation and real protocol execution, since it naturally holds due to the reasons given above. In our simulation, we omit the case where all parties are honest since the simulator simply has to follow the protocol description. In the case of three protocol participants, we provide a simulation for all cases where two parties are corrupted and one party is honest because these cases cover also all cases where just one party is corrupted. In other

words, the case where two parties are honest is a combination of cases where each of these parties is honest individually.

Since the functionality $\mathcal{F}_{preL}$ incorporates, $\mathcal{F}_L$, we refer at some point of our simulation to the simulator code for ledger channels.

We note that the indistinguishability of the simulated transcript and the transcript of the real protocol can only hold if the security properties of the underlying adaptor signature scheme hold. Namely, we require the adaptor signature scheme to fulfill the unforgeability, witness extractability and adaptability properties.

Simulator for Wrapper protocol

Pre-Creat

Case A honest and B corrupted

1. Upon A sending $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}) \xhookrightarrow{\tau_0} \mathcal{F}_{preL}$ set $T_1 = 2$ and do the following:
2. If $\mathsf{tx}^{\mathsf{f}}.\mathsf{output}[i].\mathsf{cash} \neq \gamma.\mathsf{cash}$, then ignore the message.
3. Set $\mathsf{id} := \gamma.\mathsf{id}$, generate $(R_A, r_A) \leftarrow \mathsf{GenR}$, $(Y_A, y_A) \leftarrow \mathsf{GenR}$ and send $(\text{createInfo}, \mathsf{id}, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}, R_A, Y_A) \xhookrightarrow{\tau_0} B$.
4. If $(\text{createInfo}, \mathsf{id}, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl}, R_B, Y_B) \xhookleftarrow{\tau_0+1} B$, create:

$$[\mathtt{TX}_{\mathsf{c}}] := \mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B, 0)$$
$$[\mathtt{TX}_{\mathsf{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \gamma.\mathsf{st})$$

   for $I_A := (\mathsf{pk}_A, R_B, Y_A)$, $I_B := (\mathsf{pk}_B, R_B, Y_B)$. Else stop.
5. Compute $s_{\mathsf{c}}^A \leftarrow \mathsf{pSign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}], Y_B)$, $s_{\mathsf{s}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{s}}])$ and send $(\text{createCom}, \mathsf{id}, s_{\mathsf{c}}^A, s_{\mathsf{s}}^A) \xhookrightarrow{\tau_0+1} B$.
6. If $(\text{createCom}, \mathsf{id}, s_{\mathsf{c}}^B, s_{\mathsf{s}}^B) \xhookleftarrow{\tau_0+2} B$, s.t. $\mathsf{pVrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{c}}], Y_A; s_{\mathsf{c}}^B) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^B) = 1$, set

$$\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \{\mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}]), \mathsf{Adapt}(s_{\mathsf{c}}^B, y_A)\})$$
$$\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \{s_{\mathsf{s}}^A, s_{\mathsf{s}}^B\})$$
$$\Gamma_{pre}^A(\gamma.\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\mathtt{TX}_{\mathsf{c}}, r_A, R_B, Y_B, s_{\mathsf{c}}^A), \mathtt{TX}_{\mathsf{s}}, t_{ofl}).$$

   and if B has not sent $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, i, t_{ofl})$ to $\mathcal{F}_{preL}$ send this message on behalf of B.

Pre-Update

Let $T_1 = 2$ and $T_2 = 1$ and let $|\mathsf{tid}| = 1$.

Case A is honest and B is corrupted

Upon A sending $(\texttt{PRE–UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0} \mathcal{F}_{preL}$, proceed as follows:

1. Generate new revocation public/secret pair $(R_P, \ r_P) \leftarrow \mathsf{GenR}$ and a new publishing public/secret pair $(Y_P, y_P) \leftarrow \mathsf{GenR}$ and send $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_A, Y_A) \xhookrightarrow{\tau_0^A} B$.

2. Upon $(\text{updateInfo}, \mathsf{id}, h_B, Y_B, s_{\mathsf{s}}^B) \xhookleftarrow{\tau_0^A+2} B$, set $t_{\mathsf{lock}} := \tau_0^A + t_{\mathsf{stp}} + 5 + \Delta + t_{\mathit{ofl}}$, extract tx^{f} from $\Gamma_{pre}^B(\mathsf{id})$ and

$$[\mathtt{TX}_{\mathsf{c}}] := \mathtt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B, t_{\mathsf{lock}})$$
$$[\mathtt{TX}_{\mathsf{s}}] := \mathtt{GenSplit}([\mathtt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta),$$

   for $I_A := (\mathsf{pk}_A, R_A, Y_A)$ and $I_B := (\mathsf{pk}_B, R_B, Y_B)$. If it holds that $\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{s}}]; s_{\mathsf{s}}^B) = 1$ continue. Else mark this execution as "failed" and stop.

3. If A sends $(\texttt{PRE–SETUP–OK}, \ \mathsf{id}) \xhookrightarrow{\tau_1^A \leq \tau_0^A+2+t_{\mathsf{stp}}} \mathcal{F}_{preL}$, compute $s_{\mathsf{c}}^A \leftarrow \mathsf{pSign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}], \ Y_B)$, $s_{\mathsf{s}}^A \leftarrow \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{s}}])$ and send the message $(\text{update–commitA}, \mathsf{id}, s_{\mathsf{c}}^A, s_{\mathsf{s}}^A) \xhookrightarrow{\tau_1^A} B$.

4. In round $\tau_1^A + 2$ distinguish the following cases:

   - If A receives $(\text{update–commitB}, \mathsf{id}, s_{\mathsf{c}}^B) \xhookleftarrow{\tau_1^A+2} B$ check if B has not sent $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_1^A+1} \mathcal{F}_{preL}$. If so send the message $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \xhookrightarrow{\tau_1^A+1} \mathcal{F}_{preL}$ on behalf of B. If $\mathsf{pVrfy}_{\mathsf{pk}_B}([\mathtt{TX}_{\mathsf{c}}], Y_A; s_{\mathsf{c}}^B) = 0$, then mark this execution as "failed" and stop.

   - If A receives $(\text{updateNotOk}, \ \mathsf{id}, \ r_B) \xhookleftarrow{\tau_1^A+2} B$, where $(R_B, \ r_B) \in R$, add $\Theta^A(\mathsf{id}) := \Theta^A(\mathsf{id}) \cup ([\mathtt{TX}_{\mathsf{c}}], r_B, Y_B, s_{\mathsf{c}}^A)$, instruct $\mathcal{F}_{preL}$ to send $(\texttt{PRE–UPDATE–REJECT}, \mathsf{id}) \hookrightarrow A$ and to stop and mark this execution as "failed" and stop.
   - Else, execute the simulator code for the procedure $\texttt{Wait–if–Register}^A(\mathsf{id})$ and stop.

5. If A sends $(\texttt{PRE–REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^A+2} \mathcal{F}_{preL}$, then parse $\Gamma_{pre}^A(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\mathtt{TX}}_{\mathsf{c}}, \bar{r}_A, \bar{R}_B, \bar{Y}_B, \bar{s}_{\mathtt{Com}}^A), \overline{\mathsf{tx}}^{\mathsf{s}})$ and update the channel space as $\Gamma_{pre}^A(\mathsf{id}) := (\gamma, \ \mathsf{tx}^{\mathsf{f}}, \ (\mathtt{TX}_{\mathsf{c}}, \ r_A, \ R_B, \ Y_B, \ s_{\mathsf{c}}^A), \mathtt{TX}_{\mathsf{s}})$, for $\mathtt{TX}_{\mathsf{s}} := ([\mathtt{TX}_{\mathsf{s}}], \ \{s_{\mathsf{s}}^A, \ s_{\mathsf{s}}^B\})$ and $\mathtt{TX}_{\mathsf{c}} := ([\mathtt{TX}_{\mathsf{c}}], \ \{\mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{c}}]), \ \mathsf{Adapt}(s_{\mathsf{c}}^B, \ y_A)\})$. Then send $(\text{revokeP}, \mathsf{id}, \bar{r}_A) \xhookrightarrow{\tau_1^A+2} B$. Else, execute the simulator code for the procedure $\texttt{Wait–if–Register}^A(\mathsf{id})$ and stop.

6. If A receives $(\text{revokeB}, \ \mathsf{id}, \ \bar{r}_B) \xhookleftarrow{\tau_1^A+4} B$, check if B has not sent $(\texttt{PRE–REVOKE}, \ \mathsf{id}) \xhookrightarrow{\tau_1^B+2} \mathcal{F}_{preL}$. If so send $(\texttt{PRE–REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_1^B+2} \mathcal{F}_L$ on behalf of B. Check if $(\bar{R}_B, \bar{r}_B) \in R$, then set

$$\Theta^B(\mathsf{id}) := \Theta^A(\mathsf{id}) \cup ([\overline{\mathtt{TX}}_{\mathsf{c}}], \bar{r}_B, \bar{Y}_B, \bar{s}_{\mathtt{Com}}^A)$$

Else execute the simulator code for the procedure
$\texttt{Wait–if–Register}^A(\mathsf{id})$ and stop.

Case B is honest and A is corrupted

Upon A sending $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, h_A) \overset{\tau_0}{\hookrightarrow} B$, send the message $(\texttt{PRE–UPDATE}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_{preL}$ on behalf of A, if A has not already sent this message. Proceed as follows:

1. Upon $(\text{updateReq}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, R_A, Y_A) \overset{\tau_0^B}{\hookleftarrow} A$, generate $(R_B, r_B) \leftarrow \mathsf{GenR}$ and $(Y_B, y_B) \leftarrow \mathsf{GenR}$.
2. Set $t_{\mathsf{lock}} := \tau_0^B + t_{\mathsf{stp}} + 4 + \Delta + t_{ofl}$, extract tx^{f} from $\Gamma^A_{pre}(\mathsf{id})$ and
$$[\texttt{TX}_{\mathsf{c}}] := \texttt{GenCommit}([\mathsf{tx}^{\mathsf{f}}], I_A, I_B, t_{\mathsf{lock}})$$
$$[\texttt{TX}_{\mathsf{s}}] := \texttt{GenSplit}([\texttt{TX}_{\mathsf{c}}].\mathsf{txid} \| 1, \theta)$$
where $I_A := (\mathsf{pk}_A, R_A, Y_A)$, $I_B := (\mathsf{pk}_B, R_B, Y_B)$.
3. Compute $s^B_{\mathsf{s}} \leftarrow \mathsf{Sign}_{\mathsf{sk}_B}([\texttt{TX}_{\mathsf{s}}])$, send $(\text{updateInfo}, \mathsf{id}, R_B, Y_B, s^B_{\mathsf{s}}) \overset{\tau_0^B}{\hookrightarrow} A$.
4. If B receives $(\text{update–commitA}, \mathsf{id}, s^A_{\mathsf{c}}, s^A_{\mathsf{s}}) \overset{\tau_1^B \le \tau_0^B + 2 + t_{\mathsf{stp}}}{\hookleftarrow} A$ then send $(\texttt{PRE–SETUP–OK}, \mathsf{id}) \overset{\tau_1^B}{\hookrightarrow} \mathcal{F}_{preL}$ on behalf of A, if A has not sent this message.
5. Check if $\mathsf{pVrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{c}}], Y_Q; s^P_{\mathsf{c}}) = 1$ and $\mathsf{Vrfy}_{\mathsf{pk}_P}([\texttt{TX}_{\mathsf{s}}]; s^P_{\mathsf{s}}) = 1$. Else mark this execution as "failed" and stop.
6. If B sends $(\texttt{PRE–UPDATE–OK}, \mathsf{id}) \overset{\tau_1^B}{\hookrightarrow} \mathcal{F}_{preL}$, then compute $s^B_{\mathsf{c}} \leftarrow \mathsf{pSign}([\texttt{TX}_{\mathsf{c}}], Y_A)$ and send $(\text{update–commitB}, \mathsf{id}, s^B_{\mathsf{c}}) \overset{\tau_1^B}{\hookrightarrow} A$. Else send $(\text{updateNotOk}, \mathsf{id}, r_B) \overset{\tau_1^B}{\hookrightarrow} A$, mark this execution as "failed" and stop.
7. Parse $\Gamma^B_{pre}(\mathsf{id})$ as $(\gamma, \mathsf{tx}^{\mathsf{f}}, (\overline{\texttt{TX}}_{\mathsf{c}}, \bar{r}_B, \bar{R}_A, \bar{Y}_A, \bar{s}^B_{\texttt{Com}}), \overline{\mathsf{tx}}^{\mathsf{s}})$. If B receives $(\text{revokeA}, \mathsf{id}, \bar{r}_A) \overset{\tau_1^B+2}{\hookleftarrow} A$, send $(\texttt{PRE–REVOKE}, \mathsf{id}) \overset{\tau_1^B+2}{\hookrightarrow} \mathcal{F}_{preL}$ on behalf of A, if A has not sent this message.

   Else if you do not receive $(\text{revokeA}, \mathsf{id}, \bar{r}_A) \overset{\tau_1^B+2}{\hookleftarrow} A$ or if $(\bar{R}_A, \bar{r}_A) \notin R$, execute the simulator code of the procedure
   $\texttt{Wait–if–Register}^B(\mathsf{id})$ and stop.
8. If B sends $(\texttt{PRE–REVOKE}, \mathsf{id}) \overset{\tau_1^B+2}{\hookrightarrow} \mathcal{F}_{preL}$, then set
$$\Theta^B(\mathsf{id}) := \Theta^B(\mathsf{id}) \cup ([\overline{\texttt{TX}}_{\mathsf{c}}], \bar{r}_A, \bar{Y}_A, \bar{s}^B_{\texttt{Com}})$$
$$\Gamma^B_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^{\mathsf{f}}, (\texttt{TX}_{\mathsf{c}}, r_B, R_A, Y_A, s^B_{\mathsf{c}}), \texttt{TX}_{\mathsf{s}}),$$

for $\mathtt{TX}_\mathsf{s} := ([\mathtt{TX}_\mathsf{s}],\ \{s_\mathsf{s}^A,\ s_\mathsf{s}^B\})$ and $\mathtt{TX}_\mathsf{c} := ([\mathtt{TX}_\mathsf{c}],\ \{\mathsf{Sign}_{\mathsf{sk}_B}([\mathtt{TX}_\mathsf{c}]),\ \mathsf{Adapt}(s_\mathsf{c}^A,\ y_B)\})$. Then $(\text{revokeB}, \mathsf{id}, \bar{r}_B) \xrightarrow{\tau_1^B+2} A$ and stop. Else, in round $\tau_1^B + 2$, execute the simulator code of the procedure $\mathtt{Wait\text{–}if\text{–}Register}^B(\mathsf{id})$ and stop.

Register

Case A honest and B corrupted

For party A in every round τ_0 do the following:

1. For each $\mathsf{id} \in \{0,1\}^*$ s.t. $\Gamma^A_{pre}(\mathsf{id}) \neq \perp$:
2. Parse $\Gamma^A_{pre}(\mathsf{id}) := (\gamma, \mathsf{tx}^\mathsf{f}, (\mathtt{TX}_\mathsf{c}, r_A, R_B, Y_B, s_\mathsf{c}^A), \mathtt{TX}_\mathsf{s}, t_{ofl}, x)$
3. If tx^f appeared on-chain in this round, then
   a) Set $\Gamma(\mathsf{id}) := (\gamma, \mathsf{tx}^\mathsf{f}, (\mathtt{TX}_\mathsf{c}, r_A, R_B, Y_B, s_\mathsf{c}^A), \mathtt{TX}_\mathsf{s})$.
   b) Set $\Gamma^A_{pre}(\mathsf{id}) := \perp$
   c) If $x = \mathtt{in\text{–}dispute}$, then execute the simulator code for $\mathtt{L\text{–}ForceClose}^A(\mathsf{id})$.

$\mathtt{Wait\text{–}if\text{–}Register}(\mathsf{id})$

Case A honest and B corrupted

Let τ_0 be the current round. Let $X := \Gamma^A_{pre}(\mathsf{id})$. Then set $\Gamma^A_{pre}(\mathsf{id}) := (X, \mathtt{in\text{–}dispute})$.

Proof of Theorems 6 and 18

We now provide a proof for Theorem 6 and Theorem 18. In our proof, we provide the code for a simulator, that simulates the protocol Π_V in the ideal world having access to the functionalities $\mathcal{L}$ and $\mathcal{F}_V$.

We note that since during our simulation, no ERROR messages are produced by the functionality, the protocol satisfies the security properties of the functionality $\mathcal{F}_V$ as mentioned in Section 6.3.2.

Simulator for creating virtual channels

creating virtual channels

Case A is honest and I are B are corrupt

Upon A sending $(\texttt{CREATE}, \gamma) \xhookrightarrow{\tau_0^A} \mathcal{F}_V$ set $T_1 = 6T + t_{\mathsf{stp}}$ proceed as follows:

1. Let $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$ and compute
$$\theta_A := \texttt{GenVChannelOutput}(\gamma, A).$$
2. Upon A sending $(\texttt{UPDATE}, \mathsf{id}_\alpha, \theta_A, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0^A} \mathcal{F}_L$ execute the simulator code of the update procedure for the generalized channels until the message $(\texttt{SETUP}, \mathsf{id}_\alpha, \mathsf{tid}_A)$ is sent by $\mathcal{F}_L$. If the execution stops send $(\text{peaceful–reject}, \mathsf{id}_\alpha) \hookrightarrow \mathcal{F}_V$.
3. Upon A receiving $(\texttt{SETUP}, \mathsf{id}_\alpha, \mathsf{tid}_A) \xhookleftarrow{\tau_1^A \leq \tau_0^A + T} \mathcal{F}_L$, execute the simulator code for `SetupVChannel` with input (γ, tid_A).
4. If this execution of `SetupVChannel` is recorded "failed" stop. Otherwise execute the simulator code of the update procedure for the generalized channels until the end. If the execution failed (I does not revoke) instruct $\mathcal{F}_V$ to $\texttt{L–ForceClose}(\mathsf{id}_\alpha)$.
5. If B or I have not sent $(\texttt{CREATE}, \gamma) \hookrightarrow \mathcal{F}_V$ send this message on their behalf.
6. Upon A receiving $(\texttt{CREATED}, \gamma) \xhookleftarrow{\tau_2^A \leq \tau_1^A + 5T} \mathcal{F}_V$, mark γ as created, i.e. update $\Gamma^A(\gamma.\mathsf{id})$ from $(\bot, x)$ to (γ, x).

Case I is honest and A, B are corrupted

Upon I sending $(\texttt{CREATE}, \gamma) \xhookrightarrow{\tau_0^I} \mathcal{F}_V$ proceed as follows:

1. Set $\mathsf{id}_\alpha = \gamma.\mathsf{subchan}(A), \mathsf{id}_\beta = \gamma.\mathsf{subchan}(B)$ and generate
$$\theta_A := \texttt{GenVChannelOutput}(\gamma, A)$$
$$\theta_B := \texttt{GenVChannelOutput}(\gamma, B)$$
2. Upon A and B sending $(\texttt{UPDATE}, \mathsf{id}_\alpha, \theta_A, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0^A} \mathcal{F}_L$ and $(\texttt{UPDATE}, \mathsf{id}_\alpha, \theta_B, t_{\mathsf{stp}}) \xhookrightarrow{\tau_0^B} \mathcal{F}_L$, execute the simulator code for the update procedure of the generalized channel functionality until the message $(\texttt{UPDATE–REQ}, \mathsf{id}_\alpha, \theta_A, t_{\mathsf{stp}}, \mathsf{tid}_A)$ and $(\texttt{UPDATE–REQ}, \mathsf{id}_\alpha, \theta_B, t_{\mathsf{stp}}, \mathsf{tid}_B)$ are sent by $\mathcal{F}_L$.
3. If in round $\tau_1^I \leq \tau_0^I + T$, I has received both $(\texttt{UPDATE–REQ}, \mathsf{id}_\alpha, \theta_A, t_{\mathsf{stp}}, \mathsf{tid}_A) \hookleftarrow \mathcal{F}_L$ and $(\texttt{UPDATE–REQ}, \mathsf{id}_\beta, \theta_B, t_{\mathsf{stp}}, \mathsf{tid}_B) \hookleftarrow \mathcal{F}_L$, then execute the simulator code of `SetupVChannel` with inputs $(\gamma, \mathsf{tid}_A, \mathsf{tid}_B)$. Or send $(\text{peaceful–reject}, \mathsf{id}_\alpha) \hookrightarrow \mathcal{F}_V$ and $(\text{peaceful–reject}, \mathsf{id}_\beta) \hookrightarrow \mathcal{F}_V$ if instructed by $\mathcal{E}$. Else stop.

4. If in round $\tau_2^I \leq \tau_1^I + t_{\mathsf{stp}} + T$, I receives both $(\texttt{SETUP–OK}, \mathsf{id}_\alpha)) \hookleftarrow \mathcal{F}_L$ and $(\texttt{SETUP–OK}, \mathsf{id}_\beta)) \hookleftarrow \mathcal{F}_L$, continue executing the simulator code of the update procedure of generalized channels until the messages $(\texttt{REVOKE–REQ}, \mathsf{id}_\alpha)$ and $(\texttt{REVOKE–REQ}, \mathsf{id}_\alpha)$ are sent by $\mathcal{F}_L$. Otherwise stop.
5. If in round $\tau_3^I \leq \tau_2^I + 4T$ you have received both $(\texttt{REVOKE–REQ}, \mathsf{id}_\alpha) \hookleftarrow \mathcal{F}_L$ and $(\texttt{REVOKE–REQ}, \mathsf{id}_\beta) \hookleftarrow \mathcal{F}_L$, continue executing the simulator code of the update procedure of generalized channels until the end. Otherwise stop.
6. If A or B have not sent $(\texttt{CREATE}, \gamma) \hookrightarrow \mathcal{F}_V$ send this message on their behalf. Update $\Gamma^I(\gamma.\mathsf{id})$ from $(\bot, x)$ to (γ, x).

`SetupVChannel` for channels without validity

Case A is honest and I, B are corrupted

1. Create the body of the funding transaction:

$$\begin{aligned}\mathsf{tx}_\mathsf{f}^\gamma.\mathsf{input} :=& (\mathsf{tid}_A, \mathsf{tid}_B)\\ \mathsf{tx}_\mathsf{f}^\gamma.\mathsf{output} :=& ((\gamma.\mathsf{cash}, \texttt{Multi–Sig}_{\{\gamma.\mathsf{users}\}}),\\ & (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_{\mathsf{pk}_I}))\end{aligned}$$

2. Upon A sending $(\texttt{PRE–CREATE}, \gamma, \mathsf{tx}^\mathsf{f}, 1, t_{ofl}) \overset{t_0}{\hookrightarrow} \mathcal{F}_{preL}$ where $t_{ofl} = 2T + 8\Delta$, execute the simulator code for the Pre-Create procedure of the $\mathcal{F}_{preL}$ functionality.
3. Upon A receiving $(\texttt{PRE–CREATED}, \gamma.\mathsf{id}) \overset{\tau_1 \leq \tau_0 + T}{\hookleftarrow} \mathcal{F}_{preL}$ then sign the funding transaction, i.e. $s_\mathsf{f}^B \leftarrow \mathsf{Sign}_{\mathsf{sk}_B}([\mathsf{tx}_\mathsf{f}^\gamma])$ and send $(\mathsf{createFund}, \gamma.\mathsf{id}, s_\mathsf{f}^A, [\mathsf{tx}_\mathsf{f}^\gamma]) \overset{\tau_1}{\hookrightarrow} I$. Else record this execution as "failed" and stop.
4. Upon receiving $(\mathsf{createFund}, \gamma.\mathsf{id}, s_\mathsf{f}^B, s_\mathsf{f}^I) \overset{\tau_1 + 1}{\hookleftarrow} I$, verify all signatures, i.e. check:

$$\begin{aligned}\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathsf{tx}_\mathsf{f}^\gamma]; s_\mathsf{f}^B) &= 1\\ \mathsf{Vrfy}_{\mathsf{pk}_I}([\mathsf{tx}_\mathsf{f}^\gamma], s_\mathsf{f}^I) &= 1.\end{aligned}$$

   If all checks pass define

$$\mathsf{tx}_\mathsf{f}^\gamma := \{([\mathsf{tx}_\mathsf{f}^\gamma], s_\mathsf{f}^A, s_\mathsf{f}^B, s_\mathsf{f}^I)\},$$

   and set

$$\Gamma^A(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_\mathsf{f}^\gamma, \mathsf{tid}_A)$$

   and consider procedure successfully completed. Else record this execution as "failed" and stop.

Case I is honest and A, B are corrupted

5. If I receives $(\text{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^A, [\mathsf{tx}_{\mathsf{f}}^\gamma]) \xleftarrow{\tau_2 \leq \tau_0 + T + 1} A$ and $(\text{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^B, [\mathsf{tx}_{\mathsf{f}}^\gamma]) \xleftarrow{\tau_2} B$, verify the funding transaction and signatures of A and B, i.e. check:

$$\begin{aligned}
\mathsf{Vrfy}_{\mathsf{pk}_A}([\mathsf{tx}_{\mathsf{f}}^\gamma]; s_{\mathsf{f}}^A) &= 1 \\
\mathsf{Vrfy}_{\mathsf{pk}_B}([\mathsf{tx}_{\mathsf{f}}^\gamma], s_{\mathsf{f}}^B) &= 1 \\
(\mathsf{tid}_A, \mathsf{tid}_B) &= \mathsf{tx}_{\mathsf{f}}^\gamma.\mathsf{input} \\
(\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One-Sig}_{\mathsf{pk}_I}) &\in \mathsf{tx}_{\mathsf{f}}^\gamma.\mathsf{output}.
\end{aligned}$$

6. If all checks pass, sign the funding transaction, i.e. compute

$$\begin{aligned}
s_{\mathsf{f}}^I &:= \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}_{\mathsf{f}}^\gamma]), \\
\mathsf{tx}_{\mathsf{f}}^\gamma &:= \{([\mathsf{tx}_{\mathsf{f}}^\gamma], s_{\mathsf{f}}^A, s_{\mathsf{f}}^B, s_{\mathsf{f}}^I)\}.
\end{aligned}$$

Store $\Gamma^I(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_{\mathsf{f}}^\gamma)$. Then send $(\text{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^B, s_{\mathsf{f}}^I) \xhookrightarrow{\tau_2} A$ and $(\text{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^A, s_{\mathsf{f}}^I) \xhookrightarrow{\tau_2} B$, and consider procedure successfully completed. Else record this execution as "failed" and stop.

`SetupVChannel` for channels With validity

Case A is honest and B, I are corrupted

1. Send $(\text{createInfo}, \gamma.\mathsf{id}, \mathsf{tid}_A) \xhookrightarrow{\tau_0} B$.
2. In round $\tau_1 = \tau_0 + 1$, create the body of the funding transaction:

$$\begin{aligned}
\mathsf{tx}_{\mathsf{f}}^\gamma.\mathsf{input} &:= (\mathsf{tid}_A) \\
\mathsf{tx}_{\mathsf{f}}^\gamma.\mathsf{output} &:= ((\gamma.\mathsf{cash}, \texttt{Multi-Sig}_{\{\gamma.\mathsf{users}\}}), \\
&\qquad (\gamma.\mathsf{fee}/2, \texttt{One-Sig}_{\mathsf{pk}_I}))
\end{aligned}$$

3. Upon A sending $(\texttt{PRE-CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, 1, t_{ofl}) \xhookrightarrow{\tau_1} \mathcal{F}_{preL}$ where $t_{ofl} = \gamma.\mathsf{val} + 3\Delta$, execute the simulator code for the Pre-Create procedure of the $\mathcal{F}_{preL}$ functionality. If A does not receive the message $(\texttt{PRE-CREATED}, \gamma.\mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1 + T} \mathcal{F}_{preL}$ then mark this execution as "failed" and stop.
4. If A receives $(\text{createFund}, \gamma.\mathsf{id}, s_{\mathsf{f}}^I) \xleftarrow{\tau_2 + 2} I$, verify the signature, i.e. check:

$$\mathsf{Vrfy}_{\mathsf{sk}_I}([\mathsf{tx}_{\mathsf{f}}^\gamma]; s_{\mathsf{f}}^I) = 1.$$

If the check passes, compute a signature on the fund transaction:

$$s_{\mathtt{f}}^{A} := \mathsf{Sign}_{\mathsf{sk}_A}([\mathsf{tx}_{\mathsf{f}}^{\gamma}]),$$
$$\mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{A}} := \{([\mathsf{tx}_{\mathsf{f}}^{\gamma}], s_{\mathtt{f}}^{I}, s_{\mathtt{f}}^{A})\}.$$

Else record this execution as "failed" and stop.

5. Set
$$\Gamma^{A}(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{A}}, \mathsf{tid}_A)$$
and consider procedure successfully completed.

Case B is honest and A, I are corrupted

1. If $(\mathsf{createInfo}, \gamma.\mathsf{id}, \mathsf{tid}_A) \xleftarrow{\tau_0+1} A$, create the body of the funding and the first commit and split transactions:
$$\begin{aligned}
\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{input} :=& (\mathsf{tid}_A)\\
\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{output} :=& ((\gamma.\mathsf{cash}, \mathtt{Multi\text{–}Sig}_{\{\gamma.\mathsf{users}\}}),\\
& (\gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I}))\\
\mathsf{tx}_{\mathsf{refund}}^{\gamma}.\mathsf{input} :=& (\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}||2, \mathsf{tid}_B)\\
\mathsf{tx}_{\mathsf{refund}}^{\gamma}.\mathsf{output} :=& (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I}).
\end{aligned}$$

   Else record this execution as "failed" and stop.

2. Upon B sending $(\mathtt{PRE\text{–}CREATE}, \gamma, \mathsf{tx}^{\mathsf{f}}, 1, t_{ofl}) \xleftarrow{\tau_1=\tau_0+1} \mathcal{F}_{preL}$ where $t_{ofl} = \gamma.\mathsf{val} + 3\Delta$, execute the simulator code for the Pre-Create procedure of the $\mathcal{F}_{preL}$ functionality. If B does not receive $(\mathtt{PRE\text{–}CREATED}, \gamma.\mathsf{id}) \xleftarrow{\tau_2 \leq \tau_1+T} \mathcal{F}_{preL}$ then mark this execution as "failed" and stop.

3. Compute a signature on the refund transaction, i.e., $s_{\mathtt{Ref}}^{B} \leftarrow \mathsf{Sign}_{\mathsf{sk}_B}([\mathsf{tx}_{\mathsf{refund}}^{\gamma}])$ and define $\mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{B}} := \{([\mathsf{tx}_{\mathsf{f}}^{\gamma}])\}$. Then, send $(\mathsf{createFund}, \gamma.\mathsf{id}, s_{\mathtt{Ref}}^{B}, [\mathsf{tx}_{\mathsf{refund}}^{\gamma}], [\mathsf{tx}_{\mathsf{f}}^{\gamma}]) \xhookrightarrow{\tau_2} I$, set
$$\Gamma^{B}(\gamma.\mathsf{id}) := (\bot, \mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{B}}, \mathsf{tid}_B)$$
and consider procedure successfully completed. Else record this execution as "failed" and stop.

Case I is honest and A, B are corrupted

4. If I receives the message (createFund, $\gamma.\mathsf{id}$, $s^B_{\mathtt{Ref}}$, $[\mathsf{tx}^\gamma_{\mathsf{refund}}]$, $[\mathsf{tx}^\gamma_{\mathsf{f}}]$) $\xleftarrow{\tau_3 \le \tau_0 + T + 2}$ B, verify the fund and refund transactions and signature of B, i.e. check:

$$\begin{aligned}
&\mathsf{Vrfy}_{\mathsf{sk}_B}([\mathsf{tx}^\gamma_{\mathsf{refund}}]; s^B_{\mathtt{Ref}}) = 1.\\
&[\mathsf{tx}^\gamma_{\mathsf{refund}}].\mathsf{input} = (\mathsf{tx}^\gamma_{\mathsf{f}}.\mathsf{txid}||2, \mathsf{tid}_B),\\
&[\mathsf{tx}^\gamma_{\mathsf{refund}}].\mathsf{output} = (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I}),\\
&[\mathsf{tx}^\gamma_{\mathsf{f}}].\mathsf{output}[2] = (\gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})
\end{aligned}$$

If all checks pass, sign the fund and refund transactions, i.e. compute

$$\begin{aligned}
&s^I_{\mathtt{Ref}} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}^\gamma_{\mathsf{refund}}]), s^I_{\mathsf{f}} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathsf{tx}^\gamma_{\mathsf{f}}]),\\
&\mathsf{tx}^\gamma_{\mathsf{refund}} := \{([\mathsf{tx}^\gamma_{\mathsf{refund}}], s^I_{\mathtt{Ref}}, s^B_{\mathtt{Ref}})\}.
\end{aligned}$$

Store $\Gamma^I(\gamma.\mathsf{id}) := (\bot, [\mathsf{tx}^\gamma_{\mathsf{f}}], \mathsf{tx}^\gamma_{\mathsf{refund}}, \mathsf{tid}_A, \mathsf{tid}_B)$. Then send (createFund, $\gamma.\mathsf{id}$, s^I_{f}) $\xhookrightarrow{\tau_3}$ A, and consider procedure successfully completed. Else record this execution as "failed" and stop.

Function $\mathtt{GenVChannelOutput}(\gamma, P)$

Return θ, where $\theta.\mathsf{cash} = \gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2$ and $\theta.\varphi$ is defined as follows

$$\theta.\varphi = \begin{cases}
\mathtt{Multi\text{–}Sig}_{\gamma.\mathsf{users}} \vee (\mathtt{One\text{–}Sig}_P \wedge \mathtt{CheckRelative}_{(T_p+4\Delta)}), \\
\qquad\qquad \text{if } \gamma.\mathsf{val} = \bot \\
\mathtt{Multi\text{–}Sig}_{A,I} \vee (\mathtt{One\text{–}Sig}_I \wedge \mathtt{CheckLockTime}_{\gamma.\mathsf{val}}), \\
\qquad\qquad \text{if } \gamma.\mathsf{val} \neq \bot \wedge P = A \\
\mathtt{Multi\text{–}Sig}_{B,I} \vee (\mathtt{One\text{–}Sig}_B \wedge \mathtt{CheckLockTime}_{\gamma.\mathsf{val}+2\Delta}), \\
\qquad\qquad \text{if } \gamma.\mathsf{val} \neq \bot \wedge P = B
\end{cases}$$

Simulator for updating virtual channels

Update virtual channels

Case A is honest and B is corrupt

Below we abbreviate $\mathcal{F}_{preL} := \mathcal{F}_{preL}(T_p, 1)$ and assume A is the initiating party.

1. Upon A sending ($\mathtt{UPDATE}$, id, $\overrightarrow{\theta}$, t_{stp}) $\xhookrightarrow{\tau_0^A}$ $\mathcal{F}_{preL}$, execute the simulator for the pre-update procedure of the $\mathcal{F}_{preL}$ functionality from beginning until $\mathtt{PRE\text{–}SETUP}$ is sent. If this execution is marked "failed" stop.
2. Upon A sending ($\mathtt{SETUP\text{–}OK}$, id) $\xhookrightarrow{\tau_2^A \le \tau_1^A + t_{\mathsf{stp}}}$ $\mathcal{F}_{preL}$, continue executing the simulator code until step 4. If this execution is marked "failed" stop.

3. If A does not receive $(\texttt{PRE-UPDATE-OK}, \mathsf{id}) \xleftarrow{\tau_3^A \leq \tau_2^A + T} \mathcal{F}_{preL}$ or $(\texttt{PRE-UPDATE-REJECT}, \mathsf{id}) \xleftarrow{\tau_3^A \leq \tau_2^A + T} \mathcal{F}_{preL}$, execute the simulator code for the procedure $\texttt{Offload}^A(\mathsf{id})$ and stop.
4. Upon A sending $(\texttt{PRE-REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_3^A} \mathcal{F}_{preL}$ continue executing the simulator code until the end. If this execution is marked as "failed" execute the simulator code for the procedure $\texttt{Offload}^A(\mathsf{id})$ and stop.
5. Upon A receiving $(\texttt{PRE-UPDATED}, \mathsf{id}) \xleftarrow{\tau_4^A \leq \tau_3^A + T} \mathcal{F}_{preL}$, update the channel space, i.e., let $\gamma := \Gamma^A(\mathsf{id})$, set $\gamma.\mathsf{st} := \overrightarrow{\theta}$ and $\Gamma(\mathsf{id}) := \gamma$. Else if this execution is marked as "failed" execute the simulator code for the procedure $\texttt{Offload}^A(\mathsf{id})$ and stop.

Case B is honest and A is corrupt

1. Let τ_0^B be the round in which B receives $(\texttt{PRE-UPDATE-REQ}, \mathsf{id}, \overrightarrow{\theta}, t_{\mathsf{stp}}, \mathsf{tid}) \xleftarrow{\tau_0^B} \mathcal{F}_{preL}$.
2. Let $\tau_1^B \leq \tau_0 + t_{\mathsf{stp}} + T$ be the round in which B receives the message $(\texttt{PRE-SETUP-OK}, \mathsf{id})$ $\xleftarrow{\tau_1^B \leq \tau_0 + t_{\mathsf{stp}} + T} \mathcal{F}_{preL}$.
3. Upon B sending $(\texttt{UPDATE-OK}, \mathsf{id}) \xleftarrow{\tau_1^B} \mathcal{F}_{preL}$ execute the simulator code of the pre-update procedure for the $\mathcal{F}_{preL}$ functionality until the message $\texttt{PRE-REVOKE-REQ}$ is sent by the functionality and let the this round be $\tau_2^B \leq \tau_1^B + T$. If this execution is marked as "failed" execute the simulator code of the procedure $\texttt{Offload}^B(\mathsf{id})$ and stop.
4. Upon B sending $(\texttt{PRE-REVOKE}, \mathsf{id}) \xhookrightarrow{\tau_2^B} \mathcal{F}_{preL}$ continue executing the simulator code until the end. If this execution is marked as "failed" execute the simulator code for the procedure $\texttt{Offload}^B(\mathsf{id})$ and stop.
5. Upon B receiving $(\texttt{PRE-UPDATED}, \mathsf{id}) \xleftarrow{\tau_3^B \leq \tau_2^B + T} \mathcal{F}_{preL}$, update the channel space, i.e., let $\gamma := \Gamma^B(\mathsf{id})$, set $\gamma.\mathsf{st} := \overrightarrow{\theta}$ and $\Gamma(\mathsf{id}) := \gamma$.

Simulator for offloading virtual channels

Offloading virtual channels without validity

Case A honest and I, B corrupted

1. Extract γ and $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^A(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_\mathsf{f}^\gamma$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$. Upon A sending $(\texttt{CLOSE}, \mathsf{id}_\alpha) \xhookrightarrow{\tau_0} \mathcal{F}_L$ execute the simulator code for the close procedure of generalized ledger channels.
2. If A receives $(\texttt{CLOSED}, \mathsf{id}_\alpha) \xleftarrow{\tau_1 \leq \tau_0 + T + 3\Delta} \mathcal{F}_L$, check that a transaction with tid_A appeared on $\mathcal{L}$. Else stop.
3. Let $T_2 := \tau_1 + T + 3\Delta$ and distinguish:
   - If in round $\tau_2 \leq T_2$ a transaction with tid_B appeared on $\mathcal{L}$, then $(\mathsf{post}, \mathsf{tx}_\mathsf{f}^\gamma) \xhookrightarrow{\tau_2} \mathcal{L}$.

- Else in round T_2 create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_A$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash}+\gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_A})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{pun}}])$. Then $(\mathrm{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{T_2}{\hookrightarrow} \mathcal{L}$.

4. Let $T_3 := \tau_2 + \Delta$ and distinguish the following two cases:
   - The transaction $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then update $\Gamma^A := \mathtt{ToLedgerChannel}(\Gamma^A, \gamma.\mathsf{id})$ and set $m :=$ offloaded.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then define $\Gamma^A(\gamma.\mathsf{id}) = \bot$ and set $m :=$ punished.
5. Return m in round τ_3.

Case I honest and A, B corrupted

1. Extract γ and $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^I(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_{\mathsf{f}}^{\gamma}$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ Upon I sending the messages $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ and $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code of the close procedure for the generalized channels.
2. If I receives both $(m, \mathsf{id}_\alpha) \overset{\tau_1^A \leq \tau_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_L$ and $(\mathtt{CLOSED}, \mathsf{id}_\beta) \overset{\tau_1^B \leq \tau_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_L$, check that a transaction with tid_A and a transaction with tid_B appeared on $\mathcal{L}$. Then publish $(\mathrm{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma}) \overset{\tau_1}{\hookrightarrow} \mathcal{L}$, where $\tau_1 := \max\{\tau_1^A, \tau_1^B\}$. Otherwise set $\Gamma^I(\mathsf{id}) = \bot$ and stop.
3. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $\tau_2 \leq \tau_1 + \Delta$, then $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and return "offloaded".

Offloading virtual channels with validity

Case A honest and B, I corrupted

1. Extract γ, tid_A and $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ from $\Gamma^A(\mathsf{id})$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$ and Upon A sending $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code of the close procedure of the generalized ledger channel.
2. If A receives $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{\tau_1 \leq \tau_0+T+3\Delta}{\hookleftarrow} \mathcal{F}_L$, then post $(\mathrm{post}, \mathsf{tx}_{\mathsf{f}}^{\gamma,\mathsf{A}}) \overset{\tau_2}{\hookrightarrow} \mathcal{L}$. Otherwise, set $\Gamma^A(\gamma.\mathsf{id}) = \bot$ and stop.
3. Once $\mathsf{tx}_{\mathsf{f}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $\tau_2 \leq \tau_1 + \Delta$, then set $\Gamma_L^A(\mathsf{id}) := \Gamma^A(\mathsf{id})$, $\Gamma^A(\mathsf{id}) := \bot$ and return "offloaded".

Case B honest and A, I corrupted

1. Extract γ, tid_B and $[\mathsf{tx}_{\mathsf{f}}^{\gamma}]$ from $\Gamma^B(\mathsf{id})$. Then define $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and Upon B sending $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code of the close procedure of the generalized ledger channel.

2. If B receives $(\texttt{CLOSED}, \mathsf{id}_\beta) \xleftarrow{\tau_1 \leq \tau_0 + T + 3\Delta} \mathcal{F}_L$, then continue. Otherwise, set $\Gamma^B(\gamma.\mathsf{id}) = \bot$ and stop.
3. Create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_B$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2,\ \texttt{One–Sig}_{\mathsf{pk}_B})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_B}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $\tau_2 := \max\{\tau_1, \gamma.\mathsf{val} + 2\Delta\}$ and send $(\mathsf{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{\tau_2}{\hookrightarrow} \mathcal{L}$.
4. Let $T_3 := \tau_2 + \Delta$ and distinguish the following two cases:
   - A transaction with identifier $\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then update $\Gamma_L^B(\mathsf{id}) := \Gamma^B(\mathsf{id})$ and set $m :=$ offloaded.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then define $\Gamma^B(\gamma.\mathsf{id}) = \bot$, and set $m :=$ punished.
5. Return m in round τ_3.

Case I honest and A, B corrupted

1. Extract γ, tid_A, tid_B, $\mathsf{tx}_{\mathsf{refund}}^{\gamma}$ and $[\mathsf{tx}_{\mathsf{f}}^{\gamma}]$ from $\Gamma^I(\mathsf{id})$. Then define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and upon I sending $(\texttt{CLOSE}, \mathsf{id}_\alpha) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ and $(\texttt{CLOSE}, \mathsf{id}_\beta) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code of the close procedure of the generalized ledger channel for both id_α and id_β.
2. If I receives both $(\texttt{CLOSED}, \mathsf{id}_\alpha) \xleftarrow{\tau_1^A \leq \tau_0 + T + 3\Delta} \mathcal{F}_L$ and $(\texttt{CLOSED}, \mathsf{id}_\beta) \xleftarrow{\tau_1^B \leq \tau_0 + T + 3\Delta} \mathcal{F}_L$,then continue. Otherwise, set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and stop.
3. Create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_A$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2,\ \texttt{One–Sig}_{\mathsf{pk}_I})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $\tau_2 := \max\{\tau_1^A, \gamma.\mathsf{val}\}$ and send $(\mathsf{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{\tau_2}{\hookrightarrow} \mathcal{L}$.
4. Let $T_3 := \tau_2 + \Delta$ and distinguish the following two cases:
   - A transaction with identifier $\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $\tau_3' \leq T_3$, send $(\mathsf{post}, \mathsf{tx}_{\mathsf{refund}}^{\gamma}) \overset{\tau_4}{\hookrightarrow} \mathcal{L}$ where $\tau_4 := \max\{\tau_1^B, \tau_3'\}$. Once $\mathsf{tx}_{\mathsf{refund}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $\tau_5 \leq \tau_4 + \Delta$, set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and $m :=$ offloaded.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_3'' \leq T_3$, then set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and $m :=$ punished.
5. Return m in round τ_6 where $\tau_6 := \max\{\tau_5, \tau_3''\}$.

Simulator for punishing in a virtual channel

- Upon a party sending $(\texttt{PUNISH}) \overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$, execute the simulator code for the punish procedure of the generalized channels.
- Execute the simulator code for both `Punish` and `Punish–Validity`.

Punish

Case A honest and I, B corrupted

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} = \perp$ can be extracted from $\Gamma^A(\mathsf{id})$ do the following:

1. Extract $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^A(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_\mathsf{f}^\gamma$. Check if tid_A appeared on $\mathcal{L}$. If not, then stop.
2. Denote $T_2 := \tau_1 + T + 3\Delta$ and distinguish:
   - If in round $\tau_2 \leq T_2$ the transaction with tid_B appeared on $\mathcal{L}$, then $(\text{post}, \mathsf{tx}_\mathsf{f}^\gamma) \overset{\tau_2}{\hookrightarrow} \mathcal{L}$.
   - Else in round T_2 create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_A$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \mathtt{One\text{–}Sig}_{\mathsf{pk}_A})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_A}([\mathtt{TX}_{\mathsf{pun}}])$ and $(\text{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{T_2}{\hookrightarrow} \mathcal{L}$.
3. Let $T_3 := \tau_2 + \Delta$ and distinguish the following two cases:
   - The transaction $\mathsf{tx}_\mathsf{f}^\gamma$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then update $\Gamma^A := \mathtt{ToLedgerChannel}(\Gamma^A, \gamma.\mathsf{id})$.
   - The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq T_3$, then define $\Gamma^A(\gamma.\mathsf{id}) = \perp$.

Case I honest and A, B corrupted

For every $\mathsf{id} \in \{0,1\}^*$, such that γ with $\gamma.\mathsf{val} = \perp$ can be extracted from $\Gamma^I(\mathsf{id})$ do the following:

1. Extract $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^I(\mathsf{id})$ and tid_A, tid_B from $\mathsf{tx}_\mathsf{f}^\gamma$. Check if for some $P \in \{A,\ B\}$ a transaction with identifier tid_P appeared on $\mathcal{L}$. If not, then stop.
2. Denote $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(Q)$ where $Q = \gamma.\mathsf{otherParty}(P)$ and upon I sending $(\mathtt{CLOSE}, \mathsf{id}_\alpha)$ $\overset{\tau_0}{\hookrightarrow} \mathcal{F}_L$ execute simulator the code for the punish procedure of the generalized virtual channels.
3. If I receives $(\mathtt{CLOSED}, \mathsf{id}_\alpha) \overset{\tau_1 \leq \tau_0 + T + 3\Delta}{\hookleftarrow} \mathcal{F}_L$ and tid_Q appeared on $\mathcal{L}$, $(\text{post}, \mathsf{tx}_\mathsf{f}^\gamma) \overset{\tau_1}{\hookrightarrow} \mathcal{L}$. Otherwise set $\Gamma^I(\mathsf{id}) = \perp$ and stop.
4. Once $\mathsf{tx}_\mathsf{f}^\gamma$ is accepted by $\mathcal{L}$ in round τ_2, such that $\tau_2 \leq \tau_1 + \Delta$, set $\Gamma^I(\mathsf{id}) = \perp$.

Punish–Validity

Case A honest and I, B corrupted

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} \neq \perp$ can be extracted from $\Gamma^A(\mathsf{id})$ do the following:

1. Extract tid_A and $\mathsf{tx}_\mathsf{f}^\gamma$ from $\Gamma^A(\mathsf{id})$. Check if tid_A appeared on $\mathcal{L}$. If not, then stop.
2. Send $(\text{post}, \mathsf{tx}_\mathsf{f}^\gamma) \overset{\tau_1}{\hookrightarrow} \mathcal{L}$.
3. Once $\mathsf{tx}_\mathsf{f}^\gamma$ is accepted by $\mathcal{L}$ in round $\tau_2 \leq \tau_1 + \Delta$, set $\Gamma_L^A(\mathsf{id}) := \Gamma^A(\mathsf{id})$, $\Gamma^A(\mathsf{id}) := \bot$.

Case B honest and A, I corrupted

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} \neq \bot$ can be extracted from $\Gamma^B(\mathsf{id})$ do the following:

1. Extract tid_B and $[\mathsf{tx}_\mathsf{f}^\gamma]$ from $\Gamma^B(\mathsf{id})$. Check if tid_B or $[\mathsf{tx}_\mathsf{f}^\gamma].\mathsf{txid}$ appeared on $\mathcal{L}$. If not, then stop.
2. If $\mathsf{tx}_\mathsf{f}^\gamma$ appeared on $\mathcal{L}$, set $\Gamma_L^B(\mathsf{id}) := \Gamma^B(\mathsf{id})$, $\Gamma^B(\mathsf{id}) := \bot$ and stop.
3. If tid_B appeared on $\mathcal{L}$, create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_B$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2,\ \mathtt{One\text{–}Sig}_{\mathsf{pk}_B})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_B}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $\tau_2 := \max\{\tau_1, \gamma.\mathsf{val} + 2\Delta\}$ and send $(\text{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{\tau_2}{\hookrightarrow} \mathcal{L}$.
4. If transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_3 \leq \tau_2 + \Delta$, then define $\Gamma^B(\gamma.\mathsf{id}) = \bot$.

Case I honest and A, B corrupted

For every $\mathsf{id} \in \{0,\ 1\}^*$, such that γ which $\gamma.\mathsf{val} \neq \bot$ can be extracted from $\Gamma^I(\mathsf{id})$ do the following:

1. Extract tid_A, tid_B, $\mathsf{tx}_{\mathsf{refund}}^\gamma$ and $[\mathsf{tx}_\mathsf{f}^\gamma]$ from $\Gamma^I(\mathsf{id})$. Check if tid_A or tid_B appeared on $\mathcal{L}$ or $\tau_1 = \gamma.\mathsf{val} - (3\Delta + T_p)$. If not, then stop.
2. Distinguish the following cases:
   - If $\tau_1 = \gamma.\mathsf{val} - (3\Delta + T_p)$, define $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$, $\mathsf{id}_\beta := \gamma.\mathsf{subchan}(B)$ and upon I sending the messages $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{\tau_1}{\hookrightarrow} \mathcal{F}_L$ and $(\mathtt{CLOSE}, \mathsf{id}_\beta) \overset{\tau_1}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code for the close procedure of the generalized channels for both channels id_α and id_β.
   - If tid_B appeared on $\mathcal{L}$, send $(\mathtt{CLOSE}, \mathsf{id}_\alpha) \overset{\tau_1}{\hookrightarrow} \mathcal{F}_L$.
3. If a transaction with identifier tid_A appeared on $\mathcal{L}$ in round $\tau_2 \leq \tau_1 + T + 3\Delta$, create the punishment transaction $\mathtt{TX}_{\mathsf{pun}}$ as $\mathtt{TX}_{\mathsf{pun}}.\mathsf{input} := \mathsf{tid}_A$, $\mathtt{TX}_{\mathsf{pun}}.\mathsf{output} := (\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2,\ \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})$ and $\mathtt{TX}_{\mathsf{pun}}.\mathsf{Witness} := \mathsf{Sign}_{\mathsf{sk}_I}([\mathtt{TX}_{\mathsf{pun}}])$. Then wait until round $\tau_3 := \max\{\tau_2, \gamma.\mathsf{val}\}$ and send $(\text{post}, \mathtt{TX}_{\mathsf{pun}}) \overset{\tau_3}{\hookrightarrow} \mathcal{L}$.
4. Distinguish the following two cases:

- The transaction $\mathsf{tx}_{\mathsf{f}}^{\gamma}.\mathsf{txid}$ was accepted by $\mathcal{L}$ in $\tau_4 \leq \tau_3 + \Delta$, send $(\text{post}, \mathsf{tx}_{\mathsf{refund}}^{\gamma})$ $\overset{\tau_5}{\hookrightarrow} \mathcal{L}$ where $\tau_5 := \max\{\gamma.\mathsf{val} + \Delta,\ \tau_4\}$. Once $\mathsf{tx}_{\mathsf{refund}}^{\gamma}$ is accepted by $\mathcal{L}$ in round $\tau_6 \leq \tau_5 + \Delta$, set $\Gamma^I(\gamma.\mathsf{id}) = \bot$ and stop.
- The transaction $\mathtt{TX}_{\mathsf{pun}}$ was accepted by $\mathcal{L}$ in $\tau_4 \leq \tau_3 + \Delta$, then set $\Gamma^I(\gamma.\mathsf{id}) = \bot$.

Simulator for Close in a virtual channel

Closing virtual channels

Case A honest and I, B corrupted

1. Upon A sending $(\mathtt{CLOSE},\ \mathsf{id}) \overset{\tau_0^A}{\hookrightarrow} \mathcal{F}_V$ or in round $\tau_0^A = \gamma.\mathsf{val} - (4\Delta + 7T)$ if $\gamma.\mathsf{val} \neq \bot$, proceed as follows:
2. Extract γ, $\mathsf{tx}_{\mathsf{f}}^{\gamma}$, tid_A from $\Gamma^A(\mathsf{id})$. Parse
$$\mathtt{TX}_{\mathsf{s}}^{\gamma}.\mathsf{output} = \left((c_A, \mathtt{One\text{–}Sig}_{\mathsf{pk}_A}), (c_B, \mathtt{One\text{–}Sig}_{\mathsf{pk}_B})\right).$$
3. Compute the new state of the channel $\mathsf{id}_\alpha := \gamma.\mathsf{subchan}(A)$ as
$$\overrightarrow{\theta}_A := \{(c_A, \mathtt{One\text{–}Sig}_{\mathsf{pk}_A}), (\gamma.\mathsf{cash} - c_A + \frac{\gamma.\mathsf{fee}}{2}, \mathtt{One\text{–}Sig}_{\mathsf{pk}_I})\}$$
Then, upon A sending $(\mathtt{UPDATE},\ \mathsf{id}_\alpha,\ \overrightarrow{\theta}_A,\ 0) \overset{\tau_0^A}{\hookrightarrow} \mathcal{F}_L$ execute the simulator code for the update procedure of generalized channels until $(\mathtt{SETUP},\ \mathsf{id}_\alpha,\ \mathsf{tid}'_{\mathbf{A}})$ is sent by $\mathcal{F}_L$. If this execution fails instruct $\mathcal{F}_V$ to execute $\mathtt{V\text{–}ForceClose}(\mathsf{id})$ and stop.
4. Upon A sending $(\mathtt{SETUP\text{–}OK}, \mathsf{id}_\alpha) \overset{\tau_1^A \leq \tau_0^A + T}{\longrightarrow} \mathcal{F}_L$ continue executing the simulator code for the update procedure of generalized channels until A receives $(\mathtt{UPDATE\text{–}OK}, \mathsf{id}_\alpha) \overset{\tau_2^A \leq \tau_1^A + 2T}{\longleftarrow}$ $\mathcal{F}_L$. If this execution fails instruct $\mathcal{F}_V$ to execute $\mathtt{V\text{–}ForceClose}(\mathsf{id})$ and stop.
5. Upon A sending $(\mathtt{REVOKE}, \mathsf{id}_\alpha) \overset{\tau_2^A}{\hookrightarrow} \mathcal{F}_L$ continue executing the simulator code for the update procedure of generalized channels until A receives $(\mathtt{UPDATED}, \mathsf{id}_\alpha) \overset{\tau_3^A \leq \tau_2^A + 2T}{\longleftarrow} \mathcal{F}_L$, then set $\Gamma^A(\mathsf{id}) := \bot$ and stop. If this execution fails instruct $\mathcal{F}_V$ to execute $\mathtt{V\text{–}ForceClose}(\mathsf{id})$ and stop.

Case I honest and A, B corrupted

1. Upon I sending $(\mathtt{CLOSE}, \mathsf{id}) \overset{\tau_0^I}{\hookrightarrow} \mathcal{F}_V$ or in round $\tau_0^A = \gamma.\mathsf{val} - (4\Delta + 7T)$ if $\gamma.\mathsf{val} \neq \bot$, proceed as follows:
2. Extract γ, $\mathsf{tx}_{\mathsf{f}}^{\gamma}$, tid_A and tid_B from $\Gamma^I(\mathsf{id})$.

3. Let $\mathsf{id}_\alpha = \gamma.\mathsf{subchan}(A), \mathsf{id}_\beta = \gamma.\mathsf{subchan}(B)$ and $c := \gamma.\mathsf{cash}$, execute the simulator code for the update procedure of generalized channels until $(\texttt{UPDATE–REQ}, \mathsf{id}_\alpha, \mathsf{tid}'_{\mathbf{A}}, \overrightarrow{\theta}_A, 0) \hookleftarrow \mathcal{F}_L$ and $(\texttt{UPDATE–REQ}, \mathsf{id}_\beta, \mathsf{tid}'_{\mathbf{B}}, \overrightarrow{\theta}_B, 0) \hookleftarrow \mathcal{F}_L$ are sent by $\mathcal{F}_L$ until round $\tau_1^I \leq \tau_0^I + T$.
4. Check that for some c_A, c_B s.t. $c_A + c_B + \gamma.\mathsf{fee} = c$ it holds

$$\overrightarrow{\theta}_A = \{(c_A, \texttt{One–Sig}_{\mathsf{pk}_A}), (c - c_A + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_{\mathsf{pk}_I})\}$$
$$\overrightarrow{\theta}_B = \{(c_B, \texttt{One–Sig}_{\mathsf{pk}_B}), (c - c_B + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_{\mathsf{pk}_I})\}$$

   If not, then stop.
5. continue executing the simulator code for the update procedure of generalized channels until I receives $(\texttt{SETUP–OK}, \mathsf{id}_\alpha) \hookleftarrow \mathcal{F}_L$ and $(\texttt{SETUP–OK}, \mathsf{id}_\beta)) \hookleftarrow \mathcal{F}_L$ until round $\tau_2^I \leq \tau_1^I + T$. Otherwise stop.
6. Upon I sending $(\texttt{UPDATE–OK}, \mathsf{id}_\beta) \overset{\tau_2^I}{\hookrightarrow} \mathcal{F}_L$ continue executing the simulator code for the update procedure of generalized channels until I receives the messages $(\texttt{REVOKE–REQ}, \mathsf{id}_\alpha) \hookleftarrow \mathcal{F}_L$ and $(\texttt{REVOKE–REQ}, \mathsf{id}_\beta) \hookleftarrow \mathcal{F}_L$ until round $\tau_3^I \leq \tau_2^I + 2T$.
7. Upon I sending $(\texttt{REVOKE}, \mathsf{id}_\alpha) \overset{\tau_3^I}{\hookrightarrow} \mathcal{F}_L$ and $(\texttt{REVOKE}, \mathsf{id}_\beta) \overset{\tau_3^I}{\hookrightarrow} \mathcal{F}_L$ executing the simulator code for the update procedure of generalized channels until the end and set $\Gamma^I(\mathsf{id}) := \bot$.

APPENDIX F

Appendix to Chapter 7

F.1 When to use virtual channels

In the state of the art on off-chain protocols, we can distinguish between generic 2-party applications and simple payments. The former require a direct channel between the parties and therefore it is interesting to compare VCs and direct PCs in this setting. In the latter, PCNs have already been shown to offer improvements over constructing a direct channel and therefore it is worth to compare VC against PCN payments. Next, we highlight use cases of VCs in these two settings.

VCs vs PCs for 2-party applications. Imagine that two arbitrary users that do not share a PC or a VC decide to execute a 2-party application between them. The first disadvantage of using a PC over a VC is that over their lifespan they would pay twice as many fees per on-chain transaction (i.e., to open and close the channel). At the current average Bitcoin transaction cost of 4100 satoshi (or 0.000041 BTC or 1.73 USD), the overall cost would be 8200 satoshi (3.46 USD).

Since VCs are currently not being used in practice, there is no fee model for them. To put the cost of opening a VC into perspective, we can compare it to payments over the PCN. Say Alice and Bob are connected by a path of payment channels that has 3 hops (we take the average shortest distance of a current LN snapshot). Taking the current average fees of the LN, and, say, an average transaction amount of 50,000 satoshi (21.10 USD), Alice and Bob could perform 1115 payments in the LN for the same fee of 8200 satoshi (3.46 USD). This means that in this example, the fees paid to intermediaries for operating a VC, i.e., opening and closing, is cheaper in terms of fees if these VC operating fees are less than the fees of 1115 LN payments.

More generally, we can compare the cost of VC versus PC as follows. We introduce x as a factor by which VCs are more expensive than PCN payments. A VC channel is cheaper if $l \cdot (\mathsf{BF} + \mathsf{RF} \cdot a) \cdot x < 2 \cdot \mathsf{TF}$ holds, where l is the number of hops in the

path between the two VC endpoints. Further, BF and RF are the two types of fees charged in PCN implementation such as the LN, where BF is a base fee charged by intermediaries for forwarding payments and RF a relative fee based on the payment amount. We compare this to the transactions fee on-chain TF, paid twice in the lifespan of a PC. For instance, taking the concrete values from the example above we can write the following: $3 \cdot (1 + 0.000029 \cdot a) \cdot x < 8200$.

Secondly, creating direct PCs on-demand for applications such as Discreet Log Contracts instead of VCs is again not scalable. Doing so would incur a continuous on-chain transaction load for opening and closing channels. This is against the purpose of PCs and PCNs, which aim at reducing the on-chain load.

Finally, and perhaps still more importantly, it is not possible to open a short-lived PC, since it requires waiting for the confirmation of the funding transaction on the blockchain, which is around 1 hour in Bitcoin. So for applications that are time-critical, direct PCs are not an option. Applications such as betting on a sports event, say, half an hour before they end are simply impossible with direct PCs.

VCs vs PCN payments. Due to the limited transaction size in Bitcoin, current Lightning channels are limited to holding 483 concurrent payments, which becomes especially critical in a micropayment setting. VCs can be used to overcome this issue. Simply, instead of a payment, an output can be used to collateralize a VC, which in turn can be used to again hold 483 payments or further VCs, effectively helping to mitigate this limitation.

In terms of fees, VCs are more desirable than payments over a PCN in the context of micropayments. This is due to the fact that in a PCN, the intermediaries charge a fee for every payment, while for a VC, the fee is charged only once. We can therefore say that a VC is cheaper, if the (simplified) inequality $l \cdot (\mathsf{BF} + \mathsf{RF} \cdot a) \cdot x < l \cdot (n \cdot \mathsf{BF} + \mathsf{RF} \cdot a)$ holds, where similar to above we use the base fee BF and relative fee RF of the LN. a is the sum of the amounts of all micropayments, n the number of micropayments, and x again the factor by which a VC is more expensive than a payment. We stress that for any given x there is a number of payments n, such that the use of VC becomes cheaper than payments over the PCN because the base fee BF is paid for each of the n micropayments in the PCN setting and only once in the VC setting.

Offline users. Routing multi-hop payments (MHPs) through the network requires active participation from the intermediaries. However, users may want to go offline and then cannot route MHPs. To still lend their capacity in a productive way and generate some fees, they can allow other nodes to build a VC over them, using watchtowers to ensure their balance.

F.1.1 Application scenario: Bootstrapping

According to a recent Lightning Network (LN) snapshot, the average number of channels per node is 7.8. This means that, on average, the bootstrapping of a newly created

node in the LN costs (rounding up) 8 transactions posted on-chain, i.e., one funding transaction per channel. Additional 8 transactions need to be posted on-chain when such channels are closed. VCs can reduce the on-chain bootstrapping cost of a new node in the LN. In particular, given that the LN is a connected component and assuming that each channel has enough capacity in both directions, one can open only one payment channel holding all the funds of the user and leverage it then to open a virtual channel to the other 7 nodes, thereby minimizing the overhead on-chain.

The results of our back-of-the-envelop calculations are shown in Table F.1. We exclude Elmo here, as it does not provide functionality to close virtual channels off-chain. Here we assume that there exist 4 intermediate channels to create each VC since the average shortest path length in our snapshot of the LN is 3.4, and take the results from Table 7.3 to count the number of transactions. These results show that VCs effectively move the on-chain overhead to the off-chain setting for bootstrapping, making the PCNs an attractive and cheap layer-2 solution: A user can use a single but expensive on-chain operation to put all its funds over a single channel to a well-connected node and then create many and cheap virtual channels to any frequent counterparties over the PCN topology. By doing that, the user additionally gains in liveness and privacy guarantees as VCs in Donner are not susceptible to the corresponding attacks by the intermediaries.

Table F.1: Bootstrapping cost comparison

	no VCs		LVPC [JLT20]		**Donner**	
on-/off-chain	on	off	on	off	on	off
connecting to the network	8	16	1	147	1	126
disconnecting honestly	8	0	1	84	1	84
disconnecting forcefully	8	0	120	0	8	0

Table F.2: Comparison to other virtual channel protocols. We denote *dispute* as the case where a party needs to enforce their VC funds or be compensated. In the UTXO case, this means offloading. * by synchronizing all channels, this time can be reduced to $\Theta(log(n))$. † for single-hop constructions n is constant, however, since the action/storage overhead/time delay is per user, we write $\Theta(n)$. ‡ This depends on using indirect/direct dispute.

	Perun [DEFM19]	GSCN [DFH18]	MPVC [DEF+19b]	BCVC-V/BCVC-NV [AME+21]	LVPC [JLT20]	Elmo [KL]	Donner
Scripting req.	Ethereum	Ethereum	Ethereum	Bitcoin	Bitcoin	Bitcoin + ANYPREVOUT	Bitcoin
Multi-hop	no	yes	yes	no	yes	yes	yes
Domino attack	no	no	no	yes	yes	yes	no
Path privacy	no	no	no	no	no	no	yes
Storage Overhead per party	$\Theta(n)^{\dagger}$	$\Theta(n)^{*}$	$\Theta(n)^{*}/\Theta(1)^{\ddagger}$	$\Theta(n)^{\dagger}$	$\Theta(n)^{*}$	$\Theta(n^3)$	$\Theta(1)$
Time-based fee model	yes	yes	yes	no/yes	no	no	yes
Unlimited lifetime	no	no	no	yes/no	no	yes	yes
Off-chain closing	yes	yes	yes	yes	yes	no	yes
Dispute: txs on-chain	1	1	1	$\Theta(n)^{\dagger}$	$\Theta(n)$	$\Theta(n)$	1
Dispute: time delay	$\Theta(n)^{\dagger}$	$\Theta(n)^{*}$	$\Theta(n)/1^{\ddagger}$	$\Theta(n)^{\dagger}$	$\Theta(n)^{*}$	$\Theta(n)^{*}$	1

F.2 Extended comparison and discussion

F.2.1 Extended comparison to the state of the art in VCs

Dziembowski et al. [DEFM19] proposed the first construction of VCs over a single intermediary. Recursive constructions [DFH18] followed up allowing for creating VCs on top of other VCs (or a pair composed of a VC and a PC), thereby supporting arbitrarily many intermediaries. Dziembowski et al. [DEF+19b, Per20] further extended the expressiveness of VCs proposing the notion of multi-party VCs, where a set of n participants build an n-party channel recursively from their pair-wise payment/virtual channels. Unfortunately, all the aforementioned constructions rely on the expressiveness of Turing-complete scripting languages such as that of Ethereum and are based on the account model instead of the Unspent Transaction Output (UTXO) model; thus, they are incompatible with many of the cryptocurrencies available today, including Bitcoin itself. Aumayr et al. [AME+21] have later shown how to design a Bitcoin-compatible VC through a carefully crafted cryptographic protocol in the UTXO model, supporting however only one intermediary.

Jourenko et al. [JLT20] have recently introduced the first Bitcoin-compatible construction over multiple intermediaries, called Lightweight Virtual Payment Channels (LVPC), where a VC over one hop is applied recursively to achieve a VC between two users separated by a path of any length. More recently, Kiayias and Litos have introduced Elmo [KL], a VC construction that does not rely on creating intermediate VCs, by instead relying on scripting functionality not present in Bitcoin, i.e., the opcode ANYPREVOUT. In Table F.2 we compare Donner to existing VC protocols, including those that rely on a Turing-complete scripting language or are limited to a single intermediary.

F.2.2 Extended discussion

Deterring the Domino attack with fees. One might think that the Domino attack could deterred by fees. I.e., intermediaries charge fees high enough to be compensated for having to close and reopen their channel, as well as having to claim the collateral put into the VC, in total at least three transactions per intermediary, in addition to the fees charged for the VC usage. It becomes clear, that this is an infeasible deterrence strategy and is in opposition to the aim of VCs to provide scalable and cheap payments: No user would pay three times an on-chain fee per intermediary for a VC. They would simply post an on-chain transaction or open a new direct PC.

Unidirectionally funded. Similar to current PCs in the Lightning Network, our VCs are only funded by U_0, whom we call the sending endpoint or sender. User U_n is the receiving endpoint or receiver and the intermediaries are $\{U_i\}_{i\in[1,n-1]}$. Even though the VC is only funded by U_0, once some money has been moved, they can use the channel also in the other direction. Moreover, if they want to have a channel funded from both endpoints, they can simply construct another channel from U_n to U_0.

Choosing the lifetime. The lifetime T is chosen by the two endpoints of the VC, depending on how long they plan to use the VC. They propose this to the intermediaries who can, based on this time and the amount they need to lock as a collateral, charge a fee. Note that this T has to be larger than the time it takes to settle the Blitz contracts, $T \geq 3\Delta + t_{\mathsf{c}}$, where Δ is an upper bound on the time it takes for a valid transaction to appear on the ledger (i.e., modeling the block delay as mentioned in Section 7.2) and t_{c} is the time it takes to close a channel. Intermediaries can prolong the lifetime if they agree and they can charge a fee based on time and amount.

Properties inherited from Blitz. The fee mechanism of Blitz can be reused here as well, i.e., the intermediary nodes forward fewer coins than they receive. Additionally, the outputs ϵ of $\mathsf{tx}^{\mathsf{vc}}$ represent a small number. Since they cannot be 0, they are the smallest possible value, one dust (546 satoshi), i.e., something that is insignificant in value to the sender. If a VC is closed (honestly) before the lifetime expires, parties do not need to wait until the lifetime expires to unlock their money. They can unlock it right away by using the *fast track* mechanism of Blitz. We refer the reader for these details to [AMSKM21]. Finally, reusing the stealth address and onion routing mechanism as in [AMSKM21] we achieve our desired privacy properties.

F.3 Operation examples

To illustrate the different operations for different VC protocols as examples, we provide the following figures. For rooted VCs, we show the construction in Figure 7.5 in Section 7.2. We further show the offload operation in Figure F.1, which coincides with the outcome of the Domino attack example in Section 7.3. For Donner, we show the full construction in Figure F.2 and the offload operation in Figure F.3

F.4 Extended background

F.4.1 Transaction graphs

In this section, we give a more in-depth explanation and example (Figure F.4) of our transaction graph notation. Rounded rectangles represent transactions, if they have a single border it means they are off-chain, with a double border on-chain. Incoming arrows to a transaction represent its inputs. The boxes within transactions denote outputs, the outgoing arrows define how an output can be spent.

More specifically, below an arrow we write who can spend the coins. This is usually a signature that verifies w.r.t. one or more public keys, which we denote as $\mathsf{OneSig}(\mathsf{pk})$ or $\mathsf{MultiSig}(\mathsf{pk}_1, \mathsf{pk}_2, ...)$. Above the arrow, we write additional conditions for how an output can be spent. This could be any script supported by the scripting language of the underlying blockchain, but in this paper, we only use relative and absolute time-locks. For the former, we write $\mathsf{RelTime}(t)$ or simply $+t$, which signifies that the output can be spent only if at least t rounds have passed since the transaction holding this output was

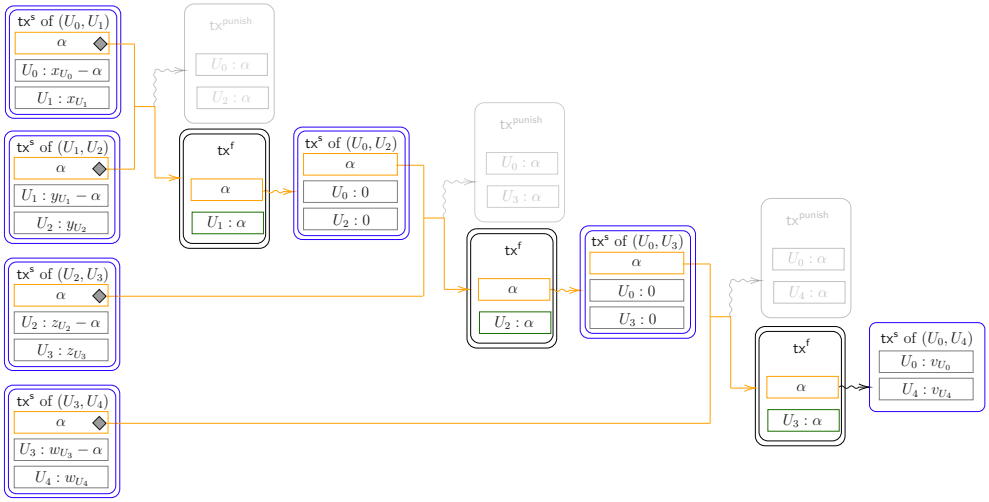

Figure F.1: Illustration showing the transactions that go on-chain in case of offloading, an operation that can be forced by a malicious enduser in the Domino attack, forcing all underlying channels to be closed.

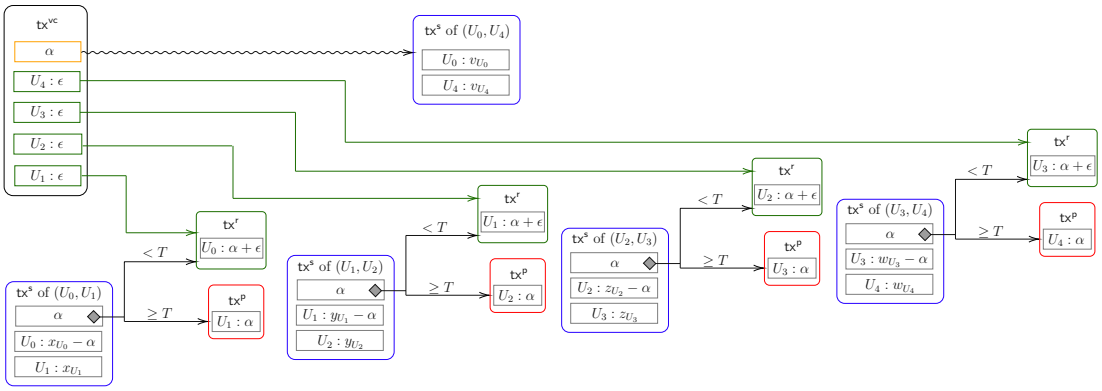

Figure F.2: Illustration of a Donner VC of U_0 and U_4 via U_1, U_2 and U_3.

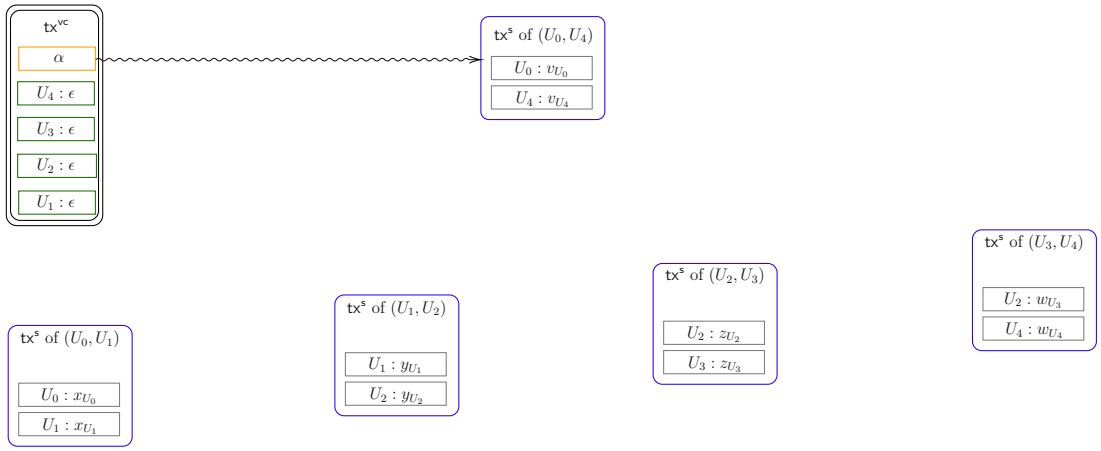

Figure F.3: Illustration of the offload operation for a Donner VC. Note that the underlying PCs remain open and only one transaction goes on-chain: $\mathsf{tx}^{\mathsf{vc}}$.

Figure F.4: (Left) Transaction tx has two outputs, one of value x_1 that can be spent by B (indicated by the gray box) with a transaction signed w.r.t. pk_B at (or after) round t_1, and one of value x_2 that can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_2 rounds passed since tx was accepted on the blockchain. (Right) Transaction tx' has one input, which is the second output of tx containing x_2 coins and has only one output, which is of value x_2 and can be spent by a transaction whose witness satisfies the output condition $\phi_1 \vee \phi_2 \vee (\phi_3 \wedge \phi_4)$. The input of tx is not shown.

accepted on the blockchain. Similarly, we write $\mathsf{AbsTime}(t)$ or simply $\geq t$ for absolute time-locks, which means that the transaction can be spent only if the blockchain is at least t blocks long. A condition can be a disjunction of subconditions $\phi = \phi_1 \vee ... \vee \phi_n$, which we denote as a diamond shape in the output box, with an outgoing arrow for each subcondition. A conjunction of subconditions is simply written as $\phi = \phi_1 \wedge ... \wedge \phi_n$.

F.4.2 Synchronization example

A multi-hop payment (MHP) allows to transfer coins from U_0 to U_n through $\{U_i\}_{i\in[1,n-1]}$ in a secure way, that is, ensuring that no intermediary is at risk of losing money. A mechanism synchronizing all channels on the path is required for a payment, such that each channel is updated to represent the fact that α coins moved from left to right. We give an example of what we mean in Figure F.5.

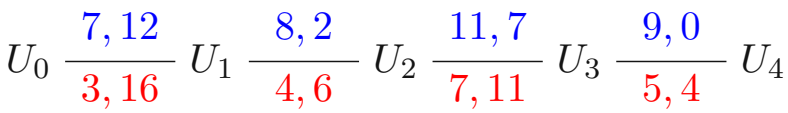

Figure F.5: Example of an MHP in a PCN. Here, U_0 pays 4 coins (disregarding any fees) to U_4, via U_1, U_2 and U_3. The lines represent payment channels. We write balances as (x, y), where x is the balance of the user on the right, and y the balance of the user on the left. Above we write the channel balances before and below after the payment. In an MHP, this change of balance should happen atomically in every channel (or not at all).

F.5 Extended macros, prerequisites and protocol

In this section, discuss the prerequisites *stealth addresses* and *onion routing*. We give extended pseudo-code for the subprocedures used in our protocol, both in the pseudocode definition given in Section 7.5 and in the formal model in Appendix F.6.3, Appendix F.6.4 and Appendix F.6.5. To be transparent about the similarities to [AMSKM21] and highlight the novelties of this work, we mark the latter in green . Further, we spell out the full protocol pseudocode, including the parts taken from. For the protocol see Figure F.8, for the two-party protocols used therein see Figure F.9. To be transparent about the similarities to [AMSKM21] and highlight the novelties of this work, we mark the latter in green color.

Subprocedures

`checkTxIn`($\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha$):

1. Check that $\mathsf{tx}^{\mathsf{in}}$ is a transaction on the ledger $\mathcal{L}$.
2. If $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\mathsf{cash} \geq n \cdot \epsilon + \alpha$ and $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0].\phi = \mathsf{OneSig}(U_0')$, that is spendable by an unused address of U_0, return $\top$. Otherwise, return $\bot$. When using this transaction (to fund $\mathsf{tx}^{\mathsf{vc}}$), the sender will pay any superfluous coins back to a fresh address of itself.

`checkChannels`($\mathsf{channelList}, U_0$):

Check that $\mathsf{channelList}$ forms a valid path from U_0 via some intermediaries to a receiver U_n and that no users are in the path twice. If not, return $\bot$. Else, return U_n.

`checkT`(n, T):

Let τ be the current round. If $T \geq \tau + n(3 + 2t_{\mathsf{u}}) + 3\Delta + t_{\mathsf{c}} + 2 + t_{\mathsf{o}}$, return $\top$. Otherwise, return $\bot$.

$\texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$:

1. Let $\mathsf{outputList} := \emptyset$ and $\mathsf{rList} := \emptyset$
2. For every channel γ_i in $\mathsf{channelList}$:
   - $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(\gamma_i.\mathsf{left}.A, \gamma_i.\mathsf{left}.B)$
   - $\mathsf{outputList} := \mathsf{outputList} \cup (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta))$
   - $\mathsf{rList} := \mathsf{rList} \cup R_i$
3. Let $\mathcal{P} := \{\gamma_i.\mathsf{left},\ \gamma_i.\mathsf{right}\}_{\gamma_i \in \mathsf{channelList}}$ and let $\mathsf{nodeList}$ be a list, where $\mathcal{P}$ is sorted from sender to receiver. Let $n := |\mathcal{P}|$.
4. Shuffle $\mathsf{outputList}$ and rList.
5. Let $\mathsf{tx}^{\mathsf{vc}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], \mathsf{outputList})$
6. Create a list $[\mathsf{msg}_i]_{i\in[0,n]}$, where $\mathsf{msg}_i := \mathcal{H}(\mathsf{tx}^{\mathsf{vc}})$
7. $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\mathsf{nodeList}, [\mathsf{msg}_i]_{i\in[0,n]})$
8. Return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$

$\texttt{genState}(\alpha_i, T, \overline{\gamma_i})$:

1. For the users $U_i := \overline{\gamma_i}.\mathsf{left} =$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$, create the output vector $\theta_i := (\theta_0, \theta_1, \theta_2)$, where
   - $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   - $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   - $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$

   where x_{U_i} and $x_{U_{i+1}}$ is the amount held by U_i and U_{i+1} in the channel, respectively.
2. Let $\mathsf{tx}_i^{\mathsf{state}}$ be a channel transaction carrying the state with $\mathsf{tx}^{\mathsf{state}}.\mathsf{output} = \theta_i$. Return $\mathsf{tx}_i^{\mathsf{state}}$.

$\texttt{checkTxEr}(U_i, a, b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i)$:

1. $x := \mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$. If $x = \perp$, return $\perp$. If U_i is the receiver and $x = \mathcal{H}(\mathsf{tx}^{\mathsf{vc}})$, return $(\top, \top, \top, \top, \top)$. Else, if $x \neq (U_{i+1}, \mathcal{H}(\mathsf{tx}^{\mathsf{vc}}), \mathsf{onion}_{i+1})$, return $\perp$.
2. For all outputs $(\mathsf{cash}, \phi) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ except output with index 0 it must hold that:
   - $\mathsf{cash} = \epsilon$
   - $\phi = \mathsf{OneSig}(\mathsf{pk}_x) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)$ for some identity pk_x
3. For exactly one output $\theta_{\epsilon_i} := (\epsilon,\ \mathsf{OneSig}(\widetilde{U_i}) \wedge \mathsf{RelTime}(t_{\mathsf{c}} + \Delta)) \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ and one element $R_i \in \mathsf{rList}$ it must hold that

- Let $\mathsf{pk}_{\widetilde{U_i}}$ be the corresponding public key of $\mathsf{OneSig}(\widetilde{U_i})$
- $\mathsf{sk}_{\widetilde{U_i}} := \mathsf{GenSk}(a, b, \mathsf{pk}_{\widetilde{U_i}}, R_i)$ must be the corresponding secret key of $\mathsf{pk}_{\widetilde{U_i}}$

4. If the checks in 2 or 3 do not hold, return $\perp$
5. Return $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$

Subprocedures used exclusively in UC model

createMaps$(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$**:**

1. For every $U_i \in \mathsf{nodeList} \setminus U_n$ do:
   - $(\mathsf{pk}_{\widetilde{U_i}}, R_i) \leftarrow \mathsf{GenPk}(U_i.A, U_i.B)$
   - $\mathsf{outputMap}(U_i) := (\epsilon, \mathsf{OneSig}(\mathsf{pk}_{\widetilde{U_i}}) \wedge \mathsf{RelTime}(t_c + \Delta))$
   - $\mathsf{rMap}(U_i) := R_i$
2. $\mathsf{rList} = \mathsf{rMap}.\texttt{values}().\texttt{shuffle}()$
3. Let $\theta_{\mathsf{vc}} := (\alpha, \mathsf{MultiSig}(U_0, U_n))$
4. $\mathsf{tx}^{\mathsf{vc}} := (\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0], [\theta_{\mathsf{vc}}, \mathsf{outputMap}.\texttt{values}()\ .\texttt{shuffle}()])$
5. Create a map $\mathsf{stealthMap}$ that stores for every user U_i that is a key in $\mathsf{outputMap}$ the corresponding output of $\mathsf{tx}^{\mathsf{vc}}$ corresponding to $\mathsf{outputMap}(U_i)$
6. Create two empty lists $\emptyset$ named $\mathsf{msgList}, \mathsf{userList}$
7. For every $U_i \in \mathsf{nodeList}$ from U_n to U_0 (in descending order):
   - Append $[\mathcal{H}(\mathsf{tx}^{\mathsf{vc}})]$ to $\mathsf{msgList}$
   - Prepend $[U_i]$ to $\mathsf{userList}$.
   - $\mathsf{onion}_i := \mathsf{CreateRoutingInfo}(\mathsf{userList}, \mathsf{msg})$
   - $\mathsf{onions}(U_i) := \mathsf{onion}_i$
8. Return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap})$

genStateOutputs$(\overline{\gamma_i}, \alpha_i, T)$**:**

1. Let $\theta'_i := \overline{\gamma_i}.\mathsf{st}$ be the current state of the channel $\overline{\gamma_i}$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ = and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. θ'_i consists of the outputs $\theta'_{U_i} := (x_{U_i}, \mathsf{OneSig}(U_i))$ and $\theta'_{U_{i+1}} := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$ holding the balances of the two users[a]. If $x_{U_i} < \alpha_i$, return $\perp$
4. Create the output vector $\theta_i := (\theta_0, \theta_1, \theta_2)$, where
   - $\theta_0 := (\alpha_i, (\mathsf{MultiSig}(U_i, U_{i+1}) \wedge \mathsf{RelTime}(T)) \vee (\mathsf{OneSig}(U_{i+1}) \wedge \mathsf{AbsTime}(T)))$
   - $\theta_1 := (x_{U_i} - \alpha_i, \mathsf{OneSig}(U_i))$
   - $\theta_2 := (x_{U_{i+1}}, \mathsf{OneSig}(U_{i+1}))$
5. Return θ_i.

genNewState$(\overline{\gamma_i}, \alpha'_i, T)$**:**

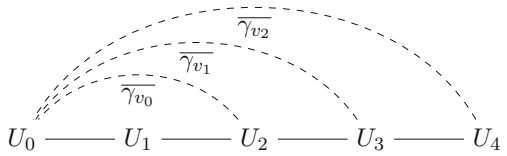

Figure F.6: Recursive virtual channel: Example A

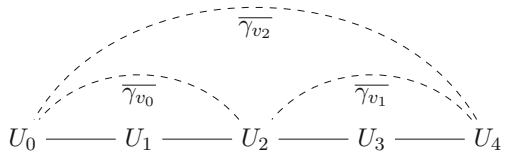

Figure F.7: Recursive virtual channel: Example B

1. Let $\theta_i := \overline{\gamma_i}.\mathsf{st}$.
2. Let $\alpha_i := \theta_i[0].\mathsf{cash}$
3. Set $\theta_0 := (\alpha_i', \theta_i[0].\phi)$
4. Set $\theta_1 := (\theta[1].\mathsf{cash} + \alpha_i - \alpha_i', \theta_i[1].\phi)$
5. Set $\theta_2 := \theta_i[2]$
6. Return vector $\theta_i' := (\theta_0, \theta_1, \theta_2)$

genRefTx$(\theta, \theta_{\epsilon_i}, U_i)$**:**

1. Create a transaction $\mathsf{tx}_i^{\mathsf{r}}$ with $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{input} := [\theta, \theta_{\epsilon_i}]$ and $\mathsf{tx}_i^{\mathsf{r}}.\mathsf{output} := (\theta.\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))$.
2. Return $\mathsf{tx}_i^{\mathsf{r}}$

genPayTx(θ, U_{i+1})**:**

1. Create a transaction $\mathsf{tx}_i^{\mathsf{p}}$ with $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{input} := [\theta]$ and $\mathsf{tx}_i^{\mathsf{p}}.\mathsf{output} := (\theta.\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$.
2. Return $\mathsf{tx}_i^{\mathsf{p}}$

[a]Possibly other outputs $\{\theta_j'\}_{j\geq 0}$ could also be present in this state. They, along with the off-chain objects there (e.g., other payments) would have to be recreated in the new state while adapting the index of the output these objects are referring to. For simplicity, we say this here in prose and omit it in the protocol, only handling the two outputs mentioned.

F.5.1 Example graphs for recursive VC

In this section, we show in Figure F.6 and Figure F.7 two example graphs that illustrate the different ways that one could recursively create a multi-hop VC using VC with a single intermediary as a building block.

F.5.2 Prerequisities

Stealth addresses. In order to hide the underlying path, we use stealth addresses [VS18] for the outputs in the transaction $\mathsf{tx}^{\mathsf{vc}}$. On a high level, every user U controls two private keys a and b. The respective public keys A and B are publicly known. A sender can use these public keys controlled by U to create a new public key P and a value R. The user U and only the user U knowing a and b can use R, P together with a and b to construct

the private key p. In particular, also the sender is unaware of p. This new one-time public key is unlinkable to U by anyone observing only R and P [VS18].

Onion routing. Like in the Lightning Network, we rely on onion routing [KBS20] techniques like Sphinx [DG09] to allow users communicate anonymously with each other. This allows users to route the VC correctly through the desired path while ensuring that intermediaries remain oblivious to the path except for their direct neighbors. On a high level, an *onion* is a layered encryption of routing information and a payload. Each user in turn can peel off one layer, revealing the next user on the path, the payload meant for the current user, and another onion, which is designated for the next user. For simplicity, we use onion routing by calling the following two functions: $\mathsf{onion} \leftarrow \mathsf{CreateRoutingInfo}(\{U_i\}_{i\in[1,n]}, \{\mathsf{msg}_i\}_{i\in[1,n]})$ generates an onion using the public keys of users $\{U_i\}_{i\in[1,n]}$ on the path, while the procedure $\mathsf{GetRoutingInfo}(\mathsf{onion}_i, U_i)$ returns the tuple $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1})$ when called by the correct user U_i, or $\perp$ otherwise.

F.6 UC modeling

For our formal security analysis, we utilize the global UC framework (GUC) [CDPW07]. In contrast to the standard Universal Composability (UC) framework, the GUC allows for a global setup, which in turn we use for modeling the blockchain as a global ledger. In this section, we go over some preliminaries and notation before presenting the ideal functionality. Note that our formal model follows closely [AEE+21, AMSKM21, DFH18, DEF+19b, AME+21, DEFM19].

F.6.1 Preliminaries, communication model and threat model

A protocol Π is executed between a set of parties $\mathcal{P}$ and runs in the presence of an adversary $\mathcal{A}$, who receives as input a security parameter $\lambda \in \mathbb{N}$ along with an auxiliary input $z \in \{0,1\}^*$. $\mathcal{A}$ can corrupt any party $P_i \in \mathcal{P}$ at the beginning of the protocol execution, i.e., a static corruption model. Corrupting a party P_i means that $\mathcal{A}$ takes control over P_i and learns its internal state. The parties and the adversary $\mathcal{A}$ take their input from the *environment* $\mathcal{E}$, a special entity that represents everything external to the protocol execution. Additionally, $\mathcal{E}$ observes the messages that are output by the parties of the protocol.

In our model, we assume a synchronous communication network with computation happening in rounds, which allows for a more natural arguing about time. This abstraction of computational rounds is formalized in the ideal functionality $\mathcal{G}_{clock}$ [KMTZ13], which represents a global clock, that proceeds to the next round if all honest parties indicate that they are ready to do so. This means that every entity always knows the given round.

Further, we assume that parties communicate via authenticated channels with guaranteed delivery after precisely one round. This is modeled by the ideal functionality $\mathcal{F}_{GDC}$: If a party P sends a message to party Q in round t, then Q receives that message in the beginning of round $t+1$ and knows that the message was sent by P. Note that

OpenVC

Setup: U_0 upon receiving $(\texttt{setup}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$, do the following:

1. Let $n := |\mathsf{channelList}|$. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0) = \bot$ or $\texttt{checkChannels}(\mathsf{channelList}, U_0) = \bot$ or $\texttt{checkT}(n, T) = \bot$, abort. Else, let $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$
2. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}) := \texttt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
3. $\overline{\gamma_{\mathsf{vc}}} := \mathsf{preCreate}\ (\mathsf{tx}^{\mathsf{vc}}, 0, U_0, U_n)$ together with U_n
4. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) :=$ $\texttt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion})$
5. $\texttt{pcSetup}(\overline{\gamma_0}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, U_1, \theta_{\epsilon_0}, \alpha_0, T)$

Open: U_{i+1} upon receiving $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i, T)$, do the following:

6. If U_{i+1} is the receiver U_n, send $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \hookrightarrow U_0$ and go idle.
7. $\texttt{pcSetup}(\overline{\gamma_{i+1}}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i - \mathsf{fee}, T)$

Finalize: U_0 upon $(\texttt{confirm}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \hookleftarrow U_n$, check that $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$ is U_n's valid signature for the transaction $\mathsf{tx}^{\mathsf{vc}}$ created in the Setup phase. If not, or if $\mathsf{tx}^{\mathsf{vc}}$ was changed, or no such confirmation was received until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{vc}}))$.

UpdateVC

Either user $U_i \in \overline{\gamma_{\mathsf{vc}}}.\mathsf{users}$ can update the virtual channel $\overline{\gamma_{\mathsf{vc}}}$ by creating a new state $\mathsf{tx}_i^{\mathsf{state}}$ and calling $\mathsf{preUpdate}(\overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}_i^{\mathsf{state}})$.

CloseVC/ProlongVC (synchronized modification)

InitClose/InitProlong

U_n: Let α_i' be the final balance of U_n in the virtual channel and $T' = T$ (Close) or let $T' > T$ be the new lifetime of the VC and leave $\alpha_i' = \alpha_i$ (Prolong). Execute $\texttt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha', T')$
U_{i-1} upon $(\top, \alpha_i', T')$: If $U_{i-1} \neq U_0$, let $\alpha_{i-1}' := \alpha_i' + \mathsf{fee}$ and $\texttt{2pModify}(\overline{\gamma_{i-2}}, \mathsf{tx}^{\mathsf{vc}}, \alpha_{i-1}', T')$

Emergency-Offload

U_0: If U_0 has not successfully performed $\texttt{2pModify}$ with the correct value α' (plus fee for each hop) until $T - t_{\mathsf{c}} - 3\Delta$, $\mathsf{publishTx}(\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0'}(\mathsf{tx}^{\mathsf{vc}}))$. Else, update $T := T'$

Respond (executed by U_i for $i \in [0, n]$ in every round)

1. If $\tau_x < T - t_{\mathsf{c}} - 2\Delta$ and $\mathsf{tx}^{\mathsf{vc}}$ on the blockchain, $\mathsf{closeChannel}(\overline{\gamma_i})$ and, after $\mathsf{tx}_i^{\mathsf{state}}$ is accepted on the blockchain within at most t_{c} rounds, wait Δ rounds. Let $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ be a signature using the secret key $\mathsf{sk}_{\widetilde{U_i}}$. $\mathsf{publishTx}(\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})))$.
2. If $\tau_x > T$, $\overline{\gamma_{i-1}}$ is closed and $\mathsf{tx}^{\mathsf{vc}}$ and $\mathsf{tx}_{i-1}^{\mathsf{state}}$ is on the blockchain, but not $\mathsf{tx}_{i-1}^{\mathsf{r}}$, $\mathsf{publishTx}(\mathsf{tx}_{i-1}^{\mathsf{p}}, (\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{p}})))$.

Figure F.8: Pseudocode of the protocol.

$\underline{\texttt{pcSetup}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_{\epsilon_i}, \alpha_i, T)\text{: (see [AMSKM21])}}$

$\underline{U_i}$

1. Let $\mathsf{tx}_i^{\mathsf{state}} := \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$ and $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_i})$
2. Send $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}})$ to U_{i+1} $(= \overline{\gamma_i}.\mathsf{right})$

$\underline{U_{i+1} \text{ upon } (\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \mathsf{tx}^{\mathsf{state}}, \mathsf{tx}_i^{\mathsf{r}}) \text{ from } U_i}$

3. Check that $\texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}) \neq \bot$, but returns some values $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$
4. Extract α_i and T from $\mathsf{tx}^{\mathsf{state}}$ and check $\mathsf{tx}_i^{\mathsf{state}} = \texttt{genState}(\alpha_i, T, \overline{\gamma_i})$
5. Check that for one output $\theta_{\epsilon_x} \in \mathsf{tx}^{\mathsf{vc}}.\mathsf{output}$ it holds that $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{GenRef}(\mathsf{tx}_i^{\mathsf{state}}, \theta_{\epsilon_x})$. If one of these previous checks failed, return $\bot$.
6. Set $\mathsf{tx}_i^{\mathsf{p}} := \mathsf{GenPay}(\mathsf{tx}_i^{\mathsf{state}})$ and send $(\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$ to U_{i+1}

$\underline{U_i \text{ upon } (\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))}$

7. If $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is not a correct signature of U_{i+1} for the $\mathsf{tx}_i^{\mathsf{r}}$ created in step 1, return $\bot$.
8. $\mathsf{updateChannel}(\overline{\gamma_i}, \mathsf{tx}_i^{\mathsf{state}})$
9. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, return $\top$. Else return $\bot$.

$\underline{U_{i+1}}$: Upon $(\mathsf{update{-}ok})$, return $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, U_{i+2}, \theta_{\epsilon_{i+1}}, \alpha_i, T)$. Else, upon $(\mathsf{update{-}fail})$, return $\bot$

$\underline{\texttt{2pModify}(\overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \alpha_i', T')}$

Let T be the timeout, α_i the amount and $\theta_{\epsilon_{i-1}}$ be the output used for the two party contract set up between U_{i-1} and U_i, known from pcSetup executed in the Open [AMSKM21] phase.

$\underline{U_i}$

1. $\mathsf{tx}_{i-1}^{\mathsf{state}'} := \texttt{genState}(\alpha_i', T', \overline{\gamma_{i-1}})$
2. $\mathsf{tx}_{i-1}^{\mathsf{r}'} := \mathsf{GenRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ //$\theta_{\epsilon_{i-1}}$ known as θ_{ϵ_x} from pcSetup
3. Send $(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))$ to U_{i-1}

$\underline{U_{i-1} \text{ upon } (\mathsf{tx}_{i-1}^{\mathsf{state}'}, \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'}))}$

1. Extract α_i' and T' from $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and check that $\alpha_i' \leq \alpha_i$, $T' \geq T$ and $\mathsf{tx}_{i-1}^{\mathsf{state}'} = \texttt{genState}(\alpha_i', T', \overline{\gamma_{i-1}})$ //α_i and T from pcSetup
2. If $U_{i-1} = U_0$, ensure that $\alpha_i' \leq x + n \cdot \mathsf{fee}$ where x is the final balance of U_n in the VC.
3. Check that $\mathsf{tx}_{i-1}^{\mathsf{r}'} = \mathsf{GenRef}(\mathsf{tx}_{i-1}^{\mathsf{state}'}, \theta_{\epsilon_{i-1}})$ //$\theta_{\epsilon_{i-1}}$ from pcSetup
4. Check that $\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})$ is a correct signature of U_i for $\mathsf{tx}_{i-1}^{\mathsf{r}'}$
5. $\mathsf{updateChannel}(\overline{\gamma_{i-1}}, \mathsf{tx}_{i-1}^{\mathsf{state}'})$
6. If, after t_{u} time has expired, the message $(\mathsf{update{-}ok})$ is returned, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$ and $\mathsf{tx}_{i-1}^{\mathsf{r}'}$, respectively. Return $(\top, \alpha_i', T')$. Else, return $\bot$.

$\underline{U_i}$: Upon $(\mathsf{update{-}ok})$, replace variables $\mathsf{tx}_{i-1}^{\mathsf{state}}$, $\mathsf{tx}_{i-1}^{\mathsf{r}}$ and $\mathsf{tx}_{i-1}^{\mathsf{p}}$ with $\mathsf{tx}_{i-1}^{\mathsf{state}'}$, $\mathsf{tx}_{i-1}^{\mathsf{r}'}$ and $\mathtt{TX}_{\mathtt{f}}^{\mathtt{i}-1} := \mathsf{GenPay}(\mathsf{tx}_{i-1}^{\mathsf{state}'})$, respectively.

Figure F.9: Protocol for 2-party channel update.

the adversary $\mathcal{A}$ is capable of reading the content of every message that is sent and can reorder messages that are sent in the same round, but cannot drop, modify, or delay messages. For a formal definition of $\mathcal{F}_{GDC}$ we refer to [DEF+19b].

In contrast to this communication between parties of $\mathcal{P}$ which takes one round, all other communication, that involves for instance the adversary $\mathcal{A}$ or the environment $\mathcal{E}$, takes zero rounds. Further, every computation that a party executes locally takes zero rounds as well.

F.6.2 Ledger and channels

We use the global ideal functionality $\mathcal{G}_{Ledger}$ to model a UTXO-based blockchain, parameterized by Δ, an upper bound on the number of rounds it takes for a valid transaction to be accepted (the blockchain delay) and a signature scheme Σ. $\mathcal{G}_{Ledger}$ communicates with a fixed set of parties $\mathcal{P}$. The environment $\mathcal{E}$ first initializes $\mathcal{G}_{Ledger}$ by setting up a key pair $(\mathsf{sk}_P, \mathsf{pk}_P)$ for every party $P \in \mathcal{P}$ and registers it to the ledger by sending $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_p)$ to $\mathcal{G}_{Ledger}$. Then, $\mathcal{E}$ sets the initial state of $\mathcal{L}$, a publicly accessible set of all published transactions. Any party $P \in \mathcal{P}$ can always post a transaction on $\mathcal{L}$ via $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}})$. If a transaction is valid, it will appear on $\mathcal{L}$ after at most Δ round, the exact number is chosen by the adversary. Recall that a transaction is valid, if all its inputs exist and are unspent, there is a correct witness for each input and a unique id.

We point out that this model is simplified: We fix the set of users instead of allowing them to join or leave dynamically. Further, transactions are in reality bundled in blocks, which are submitted by parties and $\mathcal{A}$. For a more accurate formalization, we refer to works such as [BMTZ17]. To increase readability, we opted for these simplifications.

Channels are handled by the functionality $\mathcal{F}_{Channel}$ [AME+21], which is an extension of [AEE+21] and builds on top of $\mathcal{G}_{Ledger}$. $\mathcal{F}_{Channel}$ allows to create, update, and close a payment channel between two users, as well as handling channels (pre-create and pre-update) that are funded off-chain, i.e., a virtual channel. We define t_{u} as an upper bound on rounds it takes to update and t_{c} as an upper bound on rounds it takes to close a channel (regardless of whether or not there is cooperation). We say that updating a channel takes at most t_{u} rounds and closing a channel, regardless if the parties are cooperating or not, takes at most t_{c} rounds. Finally, t_{o} is an upper bound it takes to pre-create a channel.

We assume that for our constructions, all parties in the protocol have been registered with $\mathcal{L}$, and all relevant channels between them are already open. We present an API along with an explanation of $\mathcal{F}_{Channel}$ in Figure F.10 and of $\mathcal{G}_{Ledger}$ below. For increased readability, we hide the calls to $\mathcal{G}_{clock}$ and $\mathcal{F}_{GDC}$ in our notation. Instead of explicitly calling these functionalities, we write $(\mathsf{msg}) \overset{t}{\hookrightarrow} X$ to denote sending message (msg) to X in round t and $(\mathsf{msg}) \overset{t}{\hookleftarrow} X$ to denote receiving message (msg) from X at time t. The sending/receiving entity as well as X are either a party $P \in \mathcal{P}$, the environment $\mathcal{E}$, the simulator $\mathcal{S}$, or another ideal functionality.

Interface of $\mathcal{F}_{Channel}(T, k)$ [AEE+21]

Parameters:

T: upper bound on the maximum number of consecutive off-chain communication rounds between channel users

k: number of ways the channel state can be published on the ledger

API:

Messages from $\mathcal{E}$ via a dummy user P:

- $(\texttt{sid}, \texttt{CREATE}, \overline{\gamma}, \mathsf{tid}_P) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the attribute tuple $(\overline{\gamma}.\mathsf{id}, \overline{\gamma}.\mathsf{users}, \overline{\gamma}.\mathsf{cash}, \overline{\gamma}.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\overline{\gamma}.\mathsf{users} \subset \mathcal{P}$ are the users of the channel (and $P \in \overline{\gamma}.\mathsf{users}$), $\overline{\gamma}.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is the total money in the channel and $\overline{\gamma}.\mathsf{st}$ is the initial state of the channel. tid_P defines P's input for the funding transaction of the channel. When invoked, this function asks $\overline{\gamma}.\mathsf{otherParty}$ to create a new channel.

- $(\texttt{sid}, \texttt{UPDATE}, \mathsf{id}, \theta) \overset{\tau}{\hookleftarrow} P$:
  Let $\overline{\gamma}$ be the channel where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$. When invoked by $P \in \overline{\gamma}.\mathsf{users}$ and both parties agree, the channel $\overline{\gamma}$ (if it exists) is updated to the new state θ. If the parties disagree or at least one party is dishonest, the update can fail or the channel can be forcefully closed to either the old or the new state. Regardless of the outcome, we say that t_u is the upper bound that an update takes. In the successful case, $(\texttt{sid}, \texttt{UPDATED}, \mathsf{id}, \theta) \xhookrightarrow{\leq \tau + t_\mathsf{u}} \overline{\gamma}.\mathsf{users}$ is output.

- $(\texttt{sid}, \texttt{CLOSE}, \mathsf{id}) \overset{\tau}{\hookleftarrow} P$:
  Will close the channel $\overline{\gamma}$, where $\overline{\gamma}.\mathsf{id} = \mathsf{id}$, either peacefully or forcefully. After at most t_c in round $\leq \tau + t_\mathsf{c}$, a transaction tx with the current state $\overline{\gamma}.\mathsf{st}$ as output ($\mathsf{tx}.\mathsf{output} := \overline{\gamma}.\mathsf{st}$) appears on $\mathcal{L}$ (the public ledger of $\mathcal{G}_{Ledger}$).

- $(\texttt{sid}, \texttt{PRE-CREATE}, \overline{\gamma}, \mathsf{tx}^\mathsf{f}, i, t_\mathsf{ofl}) \overset{\tau}{\hookleftarrow} P$:
  Does the same as CREATE, with the following difference. Instead of the an input for the funding transaction, the funding transaction tx^f along with an index i, defining which output of tx^f is used to fund the channel. The parameter t_ofl defines the maximum number of rounds it takes to put tx^f on-chain. If successfully invoked by both users of the channel, $\mathcal{F}_{Channel}$ returns $(\texttt{sid}, \texttt{PRE-CREATED}, \overline{\gamma}.\mathsf{id})$ after at most t_o rounds.

- $(\texttt{sid}, \texttt{PRE-UPDATE}, \mathsf{id}, \theta) \overset{\tau}{\hookleftarrow} P$:
  Does the same as UPDATE for a pre-created channel, however, in case of a dispute, $\mathcal{F}_{Channel}$ waits for tx^f to appear on the ledger within t_ofl rounds. If it does, the channel is closed.

- Additionally, $\mathcal{F}_{Channel}$ checks every round if the tx^f of a pre-created channel is put on the ledger. If it is, the pre-created channel is handled just as a normal channel from that time forward.

Figure F.10: Interface of $\mathcal{F}_{Channel}(T, k)$.

Interface of $\mathcal{G}_{Ledger}(\Delta, \Sigma)$ [AEE+21]

This functionality keeps a record of the public keys of parties. Also, all transactions that are posted (and accpeted, see below) are stored in the publicly accessible set $\mathcal{L}$ containing tuples of all accepted transactions .

Parameters:

- Δ: upper bound on the number of rounds it takes a valid transaction to be published on $\mathcal{L}$
- Σ: a digital signature scheme

API:

Messages from $\mathcal{E}$ via a dummy user $P \in \mathcal{P}$:

- $(\mathtt{sid}, \mathtt{REGISTER}, \mathsf{pk}_P) \overset{\tau}{\hookleftarrow} P$:
  This function adds an entry (pk_P, P) to PKI consisting of the public key pk_P and the user P, if it does not already exist.
- $(\mathtt{sid}, \mathtt{POST}, \overline{\mathsf{tx}}) \overset{\tau}{\hookleftarrow} P$:
  This function checks if $\overline{\mathsf{tx}}$ is a valid transaction and if yes, accepts it on $\mathcal{L}$ after at most Δ rounds.

The UC-security definition

Closely following [AMSKM21], we define Π as a *hybrid* protocol that accesses the ideal functionalities $\mathcal{F}_{\mathsf{prelim}}$ consisting of $\mathcal{F}_{Channel}$, $\mathcal{G}_{Ledger}$, $\mathcal{F}_{GDC}$ and $\mathcal{G}_{clock}$. An environment $\mathcal{E}$ that interacts with Π and an adversary $\mathcal{A}$ will on input a security parameter λ and an auxiliary input z output $\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)$. Moreover, $\phi_{\mathcal{F}_{Pay}}$ denotes the ideal protocol of ideal functionality $\mathcal{F}_{Pay}$, where the dummy users simply forward their input to $\mathcal{F}_{Pay}$. It has access to the same functionalities $\mathcal{F}_{\mathsf{prelim}}$. The output of $\phi_{\mathcal{F}_{Pay}}$ on input λ and z when interacting with $\mathcal{E}$ and a simulator $\mathcal{S}$ is denoted as $\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda, z)$.

If a protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, then any attack that is possible on the real-world protocol Π can be carried out against the ideal protocol $\phi_{\mathcal{F}_{Pay}}$ and vice versa. Our security definition is as follows.

Definition 19. A protocol Π GUC-realizes an ideal functionality $\mathcal{F}_{Pay}$, w.r.t. $\mathcal{F}_{\mathsf{prelim}}$, if for every adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that we have

$$\left\{\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\Pi,\mathcal{A},\mathcal{E}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \overset{c}{\approx} \left\{\mathtt{EXEC}^{\mathcal{F}_{\mathsf{prelim}}}_{\phi_{\mathcal{F}_{Pay}},\mathcal{S},\mathcal{E}}(\lambda, z)\right\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

where $\approx^c$ denotes computational indistinguishability.

F.6.3 Ideal functionality

In this section, we explain our ideal functionality (IF) $\mathcal{F}_{Pay}$ in prose. Note that the IF is capable of outputting an ERROR message, e.g., when a transaction does not appear

on the ledger after instructing the simulator. We remark that the only protocols that realize this IF that are of interest to us are the ones that *never* output ERROR. The cases where ERROR is output are not meaningful to us and any guarantees are lost. We use the extended macros defined in Appendix F.5. The IF is split into different parts: (i) Open-VC, (ii) Finalize-Open, (iii) Update-VC, (iv) Close-VC, (v) Emergency-Offload, and (vi) Respond. We remark on the similarity of (i), (ii), and (vi) to the IF in [AMSKM21]. To be transparent about the similarities to [AMSKM21] and highlight the novelties of this work, we mark the latter in green in the ideal functionality, formal UC protocol, and simulator.

Open-VC. This part starts with the setup phase, in which the sender U_0 invokes the IF to open a VC. In it, $\mathcal{F}_{Pay}$ takes care of creating all necessary objects, such as $\mathsf{tx}^{\mathsf{vc}}$, the onions, the stealth addresses, etc., and calls PRE-CREATE of $\mathcal{F}_{Channel}$ to set up the VC with U_n. Afterwards, $\mathcal{F}_{Pay}$ continues to do the following. If the next neighbor on the path is honest, it takes care of creating the objects and updating the channel with that neighbor, which is captured in the subprocedure Open. If the next neighbor is instead dishonest, $\mathcal{F}_{Pay}$ instructs the simulator $\mathcal{S}$ to simulate the view of the attacker. Additionally, $\mathcal{F}_{Pay}$ exposes the functionality to the simulator, which was asked to continue the open phase with a legitimate request, the simulator can perform a check to see if an id is already in use and Register to register the channel that was updated with the adversary. If the subsequent neighbor is again honest, the IF will continue handling the opening, or else the simulator will do it. This continues until the receiver U_n is reached and all channels along with their created objects are stored in the IF for each channel that contains at least one honest user. If U_n is honest, but not U_0, the last step of the Open-VC phase is actually to instruct $\mathcal{S}$ to send a confirmation to U_0. At this point, the Finalize-Open starts.

Finalize-Open. If U_0 is honest, the IF will either know that U_n completed the opening within a certain round if U_n is also honest. Or, if U_n is dishonest, $\mathcal{F}_{Pay}$ expects a confirmation from U_n via $\mathcal{S}$. If an incorrect or no confirmation was received in the correct round, the IF instructs the simulator to publish $\mathsf{tx}^{\mathsf{vc}}$, offloading the VC.

Update-VC. While the VC is open, the two endpoints can use PRE-UPDATE of $\mathcal{F}_{Channel}$ to update the VC. The IF simply forwards these messages.

Close-VC. This phase is similar to the Open-VC phase, but it is initiated by U_n, conducted from right to left, and requires fewer objects to be created. Similar to the Open-VC phase, the IF distinguishes if the left neighbor is honest or not. If it is, then $\mathcal{F}_{Pay}$ takes care of updating the channel, reducing the collateral to U_n's final balance in the VC plus its according fee. If it is dishonest, it instructs $\mathcal{S}$ to simulate the view of the adversary. If the simulator is invoked by the adversary to continue the closing with a legitimate request, the IF continues with the closure, until the sender is reached.

Emergency-Offload. If the sender of a payment is honest, the IF will expect the Close-VC request to be concluded for that payment in a certain round. If it is not, $\mathcal{F}_{Pay}$

instructs $\mathcal{S}$ to offload the VC.

Respond. This phase is executed in every round and in it, $\mathcal{F}_{Pay}$ observes if a transaction $\mathsf{tx}^{\mathsf{vc}}$ is posted on the ledger $\mathcal{L}$, which is used in channels that have an honest user and are registered as pending in the IF. If it is published early enough to refund the collateral, $\mathcal{F}_{Pay}$ closes the channels and instructs the simulator to publish the refund transaction. Else, if the lifetime of the VC T has already expired and the neighbor closes the channel, $\mathcal{F}_{Pay}$ instructs the simulator to publish the payment transaction.

Ideal Functionality $\mathcal{F}_{Pay}(\Delta)$

Parameters:

Δ : Upper bound on the time it takes a transaction to appear on $\mathcal{L}$.

Local variables:

idSet : A set of containing pairs of ids and users $(\mathtt{pid}, U_i)$ to prevent duplicate ids to avoid loops in payments.

Φ : A map, storing for a given key $(\mathtt{pid}, U_0)$ of an id $\mathtt{pid}$ and a user U_0, a tuple $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n)$, where τ_{f} is the round in which the payment confirmation is expected from the receiver, the transaction $\mathsf{tx}^{\mathsf{vc}}$ and the receiver U_n. The map is initially empty and read write access is written as $\Phi(\mathtt{pid}, U_0)$. $\Phi.\mathtt{keyList}()$ returns a set of all keys.

Γ : A set of tuples $(\mathtt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ for channels with opened payment construction, containing a payment id $\mathtt{pid}$, the channel $\overline{\gamma_i}$, the state the payment builds upon θ_i, the time T, the output used in the refund by $\overline{\gamma_i}.\mathsf{left}$ and value R_i to reconstruct the secrect key of the stealth address used. It is initially empty.

Ψ : A set of tuples $(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}})$ containing payments, that have been opened and where the receiver is honest.

t_{u}, t_{c}, t_{o} : Time it takes at most to update, close or (pre-)open a channel.

Init (executed at initialization in round t_{init}.)

Send $(\mathtt{sid}, \mathtt{init}) \xhookrightarrow{t_{\mathsf{init}}} \mathcal{S}$ and upon $(\mathtt{sid}, \mathtt{init\text{-}ok}, t_{\mathsf{u}}, t_{\mathsf{c}}, t_{\mathsf{o}}) \xhookleftarrow{t_{\mathsf{init}}} \mathcal{S}$ set t_{u}, t_{c}, t_{o} accordingly.

Open-VC

Let τ be the current round.
Setup:

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \xhookleftarrow{\tau} U_0$, if $(\mathtt{pid}, U_0) \in \mathsf{idSet}$ go idle. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\mathtt{pid}, U_0)\}$

2. Let $x := \texttt{checkChannels}(\mathsf{channelList}, U_0)$. If $x = \bot$, go idle. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in $\mathsf{channelList}$), go idle. Let $\mathsf{nodeList}$ be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\texttt{checkT}(n, T) = \bot$, go idle.
4. If $\texttt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha) = \bot$, go idle.
5. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \texttt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$.
6. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \overset{\tau}{\hookrightarrow} \mathcal{S}$ and wait 1 round.
7. Send $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau) \overset{\tau+1}{\longrightarrow} \mathcal{F}_{Channel}$
8. If not $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \overset{\tau+1+t_{\mathsf{o}}}{\longleftarrow} \mathcal{F}_{Channel}$, go idle.
9. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$.
10. Set $\Phi(\mathtt{pid}, U_0) := (\tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 2 + t_{\mathsf{o}}, \mathsf{tx}^{\mathsf{vc}}, U_n)$.
11. If U_1 honest, execute $\mathbf{Open}(\mathtt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_0, T, \overline{\gamma_0})$.
12. Else, let $\mathsf{onion}_1 := \mathsf{onions}(U_1)$ and $\theta_{\epsilon_0} := \mathsf{stealthMap}(U_0)$. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \alpha_0, T, \bot, \overline{\gamma_0}, \bot, \theta_{\epsilon_0}) \overset{\tau+1+t_{\mathsf{o}}}{\longrightarrow} \mathcal{S}$.

Continue: //Continue after a dishonest user

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. $\mathbf{Open}(\mathtt{pid}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}})$.

Check: //Sim. can check that id was not yet used

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{check\text{-}id}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. If $(\mathtt{pid}, U_i) \notin \mathsf{idSet}$, let $\mathsf{idSet} := \mathsf{idSet} \cup \{(\mathtt{pid}, U)\}$ and send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \overset{\tau}{\hookrightarrow} U_i$
3. If $(\mathtt{sid}, \mathtt{pid}, \mathtt{ACCEPT}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} U_i$, $(\mathtt{sid}, \mathtt{pid}, \mathtt{ok}, \overline{\gamma_i}) \overset{\tau}{\hookrightarrow} \mathcal{S}$.

VC-Open: //Mark VC as opened

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{payment\text{-}open}, \mathsf{tx}^{\mathsf{vc}}) \overset{\tau}{\hookleftarrow} \mathcal{S}$, let $\Psi := \Psi \cup \{(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}})\}$.

Register: //Sim. can register a channel

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{register}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. $\Gamma := \Gamma \cup \{(\mathtt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R)\}$

Open($\mathtt{pid}$, $\mathsf{nodeList}$, $\mathsf{tx}^{\mathsf{vc}}$, onions, rMap, rList, $\mathsf{stealthMap}$, α_{i-1}, T, $\overline{\gamma_{i-1}}$): Let τ be the current round and $U_i := \overline{\gamma_{i-1}}.\mathsf{right}$

1. If $(\mathtt{pid}, U_i) \in \mathsf{idSet}$, go idle.
2. $\mathsf{idSet} := \mathsf{idSet} \cup \{(\mathtt{pid}, U_i)\}$
3. If an entry after U_i in $\mathsf{nodeList}$ exists and is $\perp$, go idle.
4. If $U_i = U_n$ (i.e., last entry in $\mathsf{nodeList}$), set $U_{i+1} := \top$. Else, get U_{i+1} from $\mathsf{nodeList}$ (the entry after U_i).
5. $R_i := \mathsf{rMap}(U_i)$ and $\theta_{\epsilon_i} := \mathsf{stealthMap}(U_i)$
6. $\theta_{i-1} := \mathtt{genStateOutputs}(\overline{\gamma_{i-1}}, \alpha_{i-1}, T)$. If $\theta_{i-1} = \perp$, go idle. Else, wait 1 round.
7. $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx}^{\mathsf{er}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_{i+1}, \alpha_{i-1}, T) \xhookrightarrow{\tau+1} U_i$
8. If not $(\mathtt{sid}, \mathtt{pid}, \mathtt{ACCEPT}, \overline{\gamma_i}) \xhookleftarrow{\tau+1} U_i$, go idle. Else,wait 1 round.
9. $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_{i-1}}.\mathsf{id}, \theta_{i-1}) \xhookrightarrow{\tau+2} \mathcal{F}_{Channel}$
10. $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}) \xhookleftarrow{\tau+2+t_u} \mathcal{F}_{Channel}$, else go idle.
11. $\Gamma := \Gamma \cup (\mathtt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$
12. If $U_i = U_n$:
    - $\Psi := \Psi \cup \{(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}})\}$
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_{i-1}) \xhookrightarrow{\tau+2+t_u} U_i$
    - If U_0 is dishonest, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}) \xhookrightarrow{\tau+2+t_u} \mathcal{S}$
13. Else:
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED}) \xhookrightarrow{\tau+2+t_u} U_i$
    - If U_{i+1} honest, execute **Open**($\mathtt{pid}$, $\mathsf{nodeList}$, $\mathsf{tx}^{\mathsf{vc}}$, onions, rMap, rList, $\mathsf{stealthMap}$, $\alpha_{i-1} - \mathsf{fee}, \overline{\gamma_i}$)
    - Else, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i}) \hookrightarrow^{\tau} \mathcal{S}$, where $\mathsf{onion}_{i+1} := \mathsf{onions}(U_{i+1})$ and $\theta_{\epsilon_{i-1}} := \mathsf{stealthMap} U_{i-1}$

Finalize-Open (executed at every round)

For every $(\mathtt{pid}, U_0) \in \Phi.\mathtt{keyList}()$ do the following:

1. Let $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n) = \Phi(\mathtt{pid}, U_0)$. If for the current round τ it holds that $\tau = \tau_f$, do the following.
2. If U_n honest, check if $(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}}) \in \Psi$. If yes, let $\Psi := \Psi \setminus \{(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}})\}$ and go idle.
3. If U_n dishonest and $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmed}, \mathsf{tx}^{\mathsf{er}}_x, \sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_x)) \xhookleftarrow{\tau_{\mathsf{f}}} \mathcal{S}$, such that $\mathsf{tx}^{\mathsf{er}}_x = \mathsf{tx}^{\mathsf{vc}}$ and $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_x)$ is U_n's valid signature of $\mathsf{tx}^{\mathsf{vc}}$, go idle.

4. Send $(\texttt{sid}, \texttt{pid}, \texttt{denied}, \mathsf{tx}^{\mathsf{vc}}, U_0) \overset{\tau_f}{\hookrightarrow} \mathcal{S}$ and remove key and value for key $(\texttt{pid}, U_0)$ from Φ. $\mathsf{tx}^{\mathsf{vc}}$ must be on $\mathcal{L}$ in round $\tau' \leq \tau_f + \Delta$. Otherwise, output $(\texttt{sid}, \texttt{ERROR}) \overset{t_1}{\hookrightarrow} U_0$.

Update-VC

While VC is open, the sending and the receiving endpoint can update the VC using PRE-UPDATE of $\mathcal{F}_{Channel}$ just as they would a ledger channel.

Close-VC

Let τ be the current round.
Start:

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{SHUTDOWN}, \alpha'_{n-1}) \overset{\tau}{\hookleftarrow} U_n$, for parameter pid, fetch entry $(\texttt{pid}, \overline{\gamma_{n-1}}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ from Γ, s.t. $\overline{\gamma_{n-1}}.\mathsf{right} = U_n$. If there is no such entry, go idle.
2. Let $U_{n-1} := \overline{\gamma_{n-1}}.\mathsf{left}$.
3. If U_n is not the endpoint in VC pid, go idle.
4. If U_{n-1} honest, execute **Close**$(\texttt{pid}, \overline{\gamma_{n-1}}, \alpha'_{n-1})$
5. Else, send $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_{n-1}, \overline{\gamma_{n-1}}) \overset{\tau}{\hookrightarrow} \mathcal{S}$

Continue-Close: //Continue after a dishonest user

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{continue-close}, \overline{\gamma_{i-1}}, \alpha'_{i-1}) \overset{\tau}{\hookleftarrow} \mathcal{S}$
2. **Close**$(\texttt{pid}, \overline{\gamma_{i-1}}, \alpha'_{i-1})$.

Close$(\texttt{pid}, \overline{\gamma_i}, \alpha'_i)$: Let τ be the current round and $U_i := \overline{\gamma_i}.\mathsf{left}$

1. For the parameters pid and $\overline{\gamma_i}$, fetch entry $(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$ from Γ. If there is no entry where the parameters pid and $\overline{\gamma_i}$ match, go idle.
2. If $\overline{\gamma_i}.\mathsf{st} \neq \theta_i$, go idle.
3. Let $\alpha_i := \theta_i[0].\mathsf{cash}$. If not $0 \leq \alpha'_i \leq \alpha_i$, go idle.
4. $\theta'_i := \texttt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$. If $\theta'_i = \bot$, go idle. Else, wait 1 round.
5. $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i) \overset{\tau+1}{\longrightarrow} U_i$
6. If not $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT}) \overset{\tau+1}{\longleftarrow} U_i$, go idle.
7. $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta'_i) \overset{\tau+1}{\longrightarrow} \mathcal{F}_{Channel}$
8. If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}) \overset{\tau+1+t_u}{\longleftarrow} \mathcal{F}_{Channel}$, go idle.
9. $\Gamma := \Gamma \setminus (\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$

10. $\Gamma := \Gamma \cup (\texttt{pid}, \overline{\gamma_i}, \theta_i', \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)$
11. If $U_i = U_0$:
    - $(\texttt{sid}, \texttt{pid}, \texttt{VC-CLOSED}) \xhookrightarrow{\tau+1+t_u} U_i$
    - Remove key and value for key $(\texttt{pid}, U_0)$ from Φ.
12. Else:
    - Retrieve $\overline{\gamma_{i-1}}$ from Γ matching pid and s.t. $\overline{\gamma_{i-1}}.\mathsf{right} = U_i$
    - $(\texttt{sid}, \texttt{pid}, \texttt{CLOSED}) \xhookrightarrow{\tau+1+t_u} U_i$
    - If U_{i-1} honest, execute **Close**$(\texttt{pid}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}})$
    - Else, send $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}}) \hookrightarrow^{\tau} \mathcal{S}$.

Replace: //Update the state currently saved by the IF

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{replace}, \overline{\gamma_i}, \theta_i') \hookleftarrow^{\tau} \mathcal{S}$, let $U_i := \overline{\gamma_i}.\mathsf{left}$
2. For parameters pid and $\overline{\gamma_i}$, fetch entry $(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R) \in \Gamma$
3. $\Gamma := \Gamma \setminus \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R)\}$
4. $\Gamma := \Gamma \cup \{(\texttt{pid}, \overline{\gamma_i}, \theta_i', \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R)\}$
5. If $U_i = U_0$, remove key and value for key $(\texttt{pid}, U_0)$ from Φ.

Emergency-Offload (executed at every round)

Let τ be the current round. For every $(\texttt{pid}, U_0) \in \Phi.\texttt{keyList}()$ do the following:

1. For pid and a channel $\overline{\gamma_0}$ where $\overline{\gamma_0}.\mathsf{left} = U_0$, fetch entry $(\texttt{pid}, \overline{\gamma_0}, \theta_0, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_0}, R_0) \in \Gamma$
2. If $\tau < T - t_{\mathsf{c}} - 3\Delta$, continue with next loop iteration.
3. Else, let $(\tau_{\mathsf{f}}, \mathsf{tx}^{\mathsf{vc}}, U_n) = \Phi(\texttt{pid}, U_0)$. Send $(\texttt{sid}, \texttt{pid}, \texttt{denied}, \mathsf{tx}^{\mathsf{vc}}, U_0) \hookrightarrow^{\tau} \mathcal{S}$. $\mathsf{tx}^{\mathsf{vc}}$ must be on $\mathcal{L}$ in round $\tau' \leq \tau + \Delta$. Otherwise, output $(\texttt{sid}, \texttt{ERROR}) \hookrightarrow^{t_1} U_0$.
4. Remove key and value for key $(\texttt{pid}, U_0)$ from Φ.

Respond (executed at the end of every round)

Let t be the current round. For every element $(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i) \in \Gamma$, check if $\overline{\gamma_i}.\mathsf{st} = \theta_i$ and $\mathsf{tx}^{\mathsf{vc}}$ is on $\mathcal{L}$. If yes, do the following:

Revoke: If $\gamma_i.\mathsf{left}$ honest and $t < T - t_{\mathsf{c}} - 2\Delta$ do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)\}$.

- $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$
- At time $t + t_{\mathsf{c}}$, a transaction tx with $\mathsf{tx}.\mathsf{output} = \overline{\gamma_i}.\mathsf{st}$ has to be on $\mathcal{L}$. If not, do the following. If $(\texttt{ssid}_C, \texttt{PUNISHED}, \overline{\gamma_i}.\mathsf{id}) \overset{\tau<T}{\longleftarrow} \mathcal{F}_{Channel}$, go idle. Else, send $(\texttt{sid}, \texttt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i.\mathsf{users}$.
- Wait for Δ rounds, then $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma_i}, \theta_{\epsilon_i}, R_i) \overset{t'<T-\Delta}{\longrightarrow} \mathcal{S}$
- At time $t'' < T$, check whether a transaction tx' appears on $\mathcal{L}$ with $\mathsf{tx}'.\mathsf{input} = [\theta_{\epsilon_i}, tx.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = [(\mathsf{tx}.\mathsf{output}[0].\mathsf{cash} + \theta_{\epsilon_i}.\mathsf{cash}, \mathsf{OneSig}(U_i))]$. If it appears, send $(\texttt{sid}, \texttt{pid}, \texttt{REVOKED}) \overset{t''}{\hookrightarrow} \overline{\gamma_i}.\mathsf{left}$. If not, send $(\texttt{sid}, \texttt{ERROR}) \overset{T}{\hookrightarrow} \gamma_i.\mathsf{users}$.

Force-Pay: Else, if a transaction tx with $\mathsf{tx}.\mathsf{output} = \overline{\gamma_i}.\mathsf{st}$ is on-chain and $\mathsf{tx}.\mathsf{output}[0]$ is unspent (i.e., there is no transaction on $\mathcal{L}$, that uses is as input), $t \geq T$ and U_{i+1} is honest, do the following.

- Set $\Gamma := \Gamma \setminus \{(\texttt{pid}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R_i)\}$.
- Send $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma_i}) \overset{t}{\hookrightarrow} \mathcal{S}$
- In round $t + \Delta$ transaction tx' with $\mathsf{tx}'.\mathsf{input} = [\mathsf{tx}.\mathsf{output}[0]]$ and $\mathsf{tx}'.\mathsf{output} = (\mathsf{tx}.\mathsf{output}[0].\mathsf{cash}, \mathsf{OneSig}(U_{i+1}))$ must have appeared on $\mathcal{L}$. If yes, $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY}) \overset{t+\Delta}{\longrightarrow} \overline{\gamma_i}.\mathsf{right}$. Otherwise, $(\texttt{sid}, \texttt{ERROR}) \overset{t+\Delta}{\longrightarrow} \gamma_i.\mathsf{users}$.

F.6.4 Protocol

In this section, we give the formal protocol Π along with a short description of it. We note that for simplicity, we assume that users do not update or close the channels involved with virtual channels[1] Also, a user knows if it is an endpoint (sender/receiver) or an intermediary of a VC as well as its direct neighbors on the path. Following, the simulator simulating an honest user knows that also.

The protocol is similar to the simplified pseudo-code presented in Section 7.5. The main differences lie in having VC ids that allow handling multiple different VCs, the notion of time, and the environment $\mathcal{E}$. Briefly, the protocol starts with $\mathcal{E}$ invoking U_0 to set up the initial objects and pre-create the VC with U_n. Then U_0 asks its neighbor U_1 to exchange the necessary transactions and update their channel to hold the collateral. This is continued until the receiver U_n is reached. In the finalize phase, U_n sends a confirmation to U_0, indicating that the VC is open. In the Update VC phase, the channel can be used. The Close VC phase updates the collateral from right to left to hold U_n's final balance in the VC. The Respond phase is there, for users to react to $\mathsf{tx}^{\mathsf{vc}}$ being posted on the ledger, and triggers either a refund or claim of the collateral. We point to the similarities of Open VC, Finalize, and Respond with the formal protocol description in [AMSKM21].

[1] In reality, they can take part in multiple VCs, update, close or use their channels in some other fashion while a VC is open. For this, they recreate the output used for the collateral and $\mathsf{tx}_i^{\mathsf{r}}$, but we omit this for readability.

Protocol Π

Let $\mathsf{fee} \in \mathbb{N}$ be a system parameter known to every user.

Local variables of U_i (all initially empty):

pidSet :	A set storing every payment id pid that a user has participated in to prevent duplicates.
paySet :	A map storing tuples $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n)$ where pid is an id, τ_{f} is the round in which a confirmation is expected from the receiver U_n for the payments that have been opened by this user.
local :	A map, storing for a given pid U_i's local copy of $\mathsf{tx}^{\mathsf{vc}}$ and T in a tuple $(\mathsf{tx}^{\mathsf{vc}}, T)$.
left :	A map, storing for a given pid a tuple $(\overline{\gamma_{i-1}}, \theta_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1})$ containing channel with its left neighbor U_{i-1}, the state and the transaction $\mathsf{tx}^{\mathsf{r}}_{i-1}$ for U_i's left channel in the payment pid.
right :	A map, sotring for a given pid a tuple $(\overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U_i}})$ containing the channel with its right neighbor, the state, the transaction $\mathsf{tx}^{\mathsf{r}}_i$ and the key necessary for signing the refund transaction in the payment pid.
rightSig :	A map, storing for a given pid the signature for $\mathsf{tx}^{\mathsf{r}}_i$ of the right neighbor $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ in the payment pid.

Open VC

Setup: In every round, every node $U_0 \in \mathcal{P}$ does the following. We denote τ_0 as the current round.

U_0 upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0}) \xleftarrow{\tau_0} \mathcal{E}$

1. If pid $\in$ pidSet, abort. Add pid to pidSet.
2. Let $x :=$ checkChannels(channelList, U_0). If $x = \bot$, abort. Else, let $U_n := x$. If $\overline{\gamma_0}$ is not the full channel between U_0 and his right neighbor $U_1 := \overline{\gamma_0}.\mathsf{right}$ (corresponding to the channel skeleton γ_0 in channelList), go idle. Let nodeList be a list of all the users on the path sorted from U_0 to U_n.
3. Let $n := |\mathsf{channelList}|$. If $\mathtt{checkT}(n, T) = \bot$, abort.
4. If $\mathtt{checkTxIn}(\mathsf{tx}^{\mathsf{in}}, n, U_0, \alpha) = \bot$, abort
5. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}) := \mathtt{createMaps}(U_0, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{in}}, \alpha)$.
6. $(\mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0) := \mathtt{genTxEr}(U_0, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}})$
7. $\mathsf{paySet} := \mathsf{paySet} \cup \{(\mathtt{pid}, \tau_{\mathsf{f}} := \tau + n \cdot (2 + t_{\mathsf{u}}) + 2 + t_{\mathsf{o}}, U_n)\}$
8. $(\mathsf{sk}_{\widetilde{U_0}}, \theta_{\epsilon_0}, R_0, U_1, \mathsf{onion}_1) := \mathtt{checkTxEr}(U_0, U_0.a, U_0.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_0)$
9. Set $\mathsf{local}(\mathtt{pid}) := (\mathsf{tx}^{\mathsf{vc}}, T)$.
10. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookrightarrow{\tau_0} U_n$, wait 1 round.

11. Send $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T-\tau_0) \xleftrightarrow{\tau_0+1} \mathcal{F}_{Channel}$
12. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xleftrightarrow{\tau_0+1+t_{\mathsf{o}}} \mathcal{F}_{Channel}$, go idle.
13. Set $\alpha_0 := \alpha + \mathsf{fee} \cdot (n-1)$ and compute:
    - $\theta_0 := \texttt{genStateOutputs}(\overline{\gamma_0}, \alpha_0, T)$
    - $\mathsf{tx}^{\mathsf{r}}_0 := \texttt{genRefTx}(\theta_0, \theta_{\epsilon_0}, U_0)$
14. Set $\mathsf{right}(\texttt{pid}) := (\overline{\gamma_0}, \theta_0, \mathsf{tx}^{\mathsf{r}}_0, \mathsf{sk}_{\widetilde{U_0}})$.
15. Send $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_1, \theta_0, \mathsf{tx}^{\mathsf{r}}_0) \xleftrightarrow{\tau_0+1+t_{\mathsf{o}}} U_1$.

U_n upon $(\texttt{sid}, \texttt{pid}, \texttt{pre-create-vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xleftrightarrow{\tau} U_0$

1. $(\texttt{ssid}_C, \texttt{PRE-CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T-\tau) \xleftrightarrow{\tau} \mathcal{F}_{Channel}$
2. If not $(\texttt{ssid}_C, \texttt{PRE-CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xleftrightarrow{\tau+t_{\mathsf{o}}} \mathcal{F}_{Channel}$, mark VC as unusable.

Open: In every round, every node $U_{i+1} \in \mathcal{P}$ does the following. We denote τ_x as the current round.

U_{i+1} upon
$(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i) \xleftrightarrow{\tau_x} U_i$

1. Perform the following checks:
   - Verify that $\texttt{pid} \notin \mathsf{pidSet}$. Add $\texttt{pid}$ to pidSet
   - Let $x := \texttt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1})$.
     Check that $x \neq \perp$, but instead $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2})$.
   - Set $\alpha_i = \theta_i[0].\mathsf{cash}$ and extract T from $\theta_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$).
   - Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\theta_i = \texttt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$.
   - Check that $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\theta_i, \theta_{\epsilon_x}, U_i)$, where θ_{ϵ_x} is an output of $\mathsf{tx}^{\mathsf{vc}}$, s.t. $\theta_{\epsilon_x} \neq \theta_{\epsilon_{i+1}}$.
2. If one or more of the previous checks fail, abort. Otherwise, send $(\texttt{sid}, \texttt{pid}, \texttt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T) \xleftrightarrow{\tau_x} \mathcal{E}$.
3. If $(\texttt{sid}, \texttt{pid}, \texttt{ACCEPT}, \overline{\gamma_{i+1}}) \xleftrightarrow{\tau_x} \mathcal{E}$, generate $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$. Otherwise stop.
4. Set $\mathsf{local}(\texttt{pid}) := (\mathsf{tx}^{\mathsf{er}}_i, T)$, $\mathsf{left}(\texttt{pid}) := (\overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i)$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xleftrightarrow{\tau_x} U_i$.

U_i upon $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)) \xleftrightarrow{\tau_i+2} U_{i+1}$

(The round τ_i given U_i and pid is defined in Setup or in Open step (6), the round when the update is successful.)

5. Check that $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ is a valid signature for $\mathsf{tx}^{\mathsf{r}}_i$. If yes, set $\mathsf{rightSig}(\mathtt{pid}) := \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}}_i)$ and $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xrightarrow{\tau_i+2} \mathcal{F}_{Channel}$.

U_{i+1} upon $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xleftarrow{\tau_x+1+t_{\mathsf{u}}} \mathcal{F}_{Channel}$

6. Define $\tau_{(i+1)} := \tau_x + 1 + t_{\mathsf{u}}$.
7. If U_{i+1} is not the receiver, using the values of step 1:
   - Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED}) \xrightarrow{\tau_{i+1}} \mathcal{E}$.
   - $(\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, R_{i+1}, U_{i+2}, \mathsf{onion}_{i+2}) := \mathtt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{er}}_i, \mathsf{rList}, \mathsf{onion}_{i+1})$
   - $\theta_{i+1} := \mathtt{genStateOutputs}(\overline{\gamma_{i+1}}, \alpha_i - \mathsf{fee}, T)$
   - $\mathsf{tx}^{\mathsf{r}}_{i+1} := \mathtt{genRefTx}(\theta_{i+1}, \theta_{\epsilon_{i+1}}, U_{i+1})$
   - Set $\mathsf{right}(\mathtt{pid}) := (\overline{\gamma_{i+1}}, \theta_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}, \mathsf{sk}_{\widetilde{U_{i+1}}})$
   - Send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+2}, \theta_{i+1}, \mathsf{tx}^{\mathsf{r}}_{i+1}) \xrightarrow{\tau_{i+1}} U_{i+2}$.
8. If U_{i+1} is the receiver:
   - $\mathsf{msg} := \mathsf{GetRoutingInfo}(\mathsf{onion}_{i+1}, U_{i+1})$
   - Create the signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{er}}_i)$ as confirmation and send $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xrightarrow{\tau_{i+1}} U_0$. Send the message $(\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_i) \xrightarrow{\tau_{i+1}} \mathcal{E}$.

Finalize

U_0 in every round τ

For every entry $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n) \in \mathsf{paySet}$ do the following if $\tau = \tau_{\mathsf{f}}$:

1. Upon receiving $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xleftarrow{\tau} U_n$, continue if $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$ is a valid signature for $\mathsf{tx}^{\mathsf{vc}}$. Otherwise, go to step (3).
2. Let $(x, T) = \mathsf{local}(\mathtt{pid})$. If $x = \mathsf{tx}^{\mathsf{vc}}$, go idle. Otherwise, continue with the next step.
3. Sign $\mathsf{tx}^{\mathsf{vc}}$ yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})$ and set $\overline{\mathsf{tx}^{\mathsf{vc}}} := (\mathsf{tx}^{\mathsf{vc}}, (\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})))$. Send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \xhookrightarrow{\tau} \mathcal{G}_{Ledger}$ and remove $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n)$ from paySet.

Update VC

While VC is open, the sending and the receiving endpoint can update the VC using PRE-UPDATE of $\mathcal{F}_{Channel}$ just as they would a ledger channel.

Close VC

Shutdown: In every round, every node $U_n \in \mathcal{P}$ does the following. We denote τ_0 as the current round.

U_n upon $(\texttt{sid}, \texttt{pid}, \texttt{SHUTDOWN}, \alpha'_{n-1}) \overset{\tau_n}{\hookleftarrow} \mathcal{E}$

1. If $\texttt{pid} \notin \mathsf{pidSet}$, abort.
2. If U_n is not the receiving endpoint in the VC, abort.
3. Retrieve $(\overline{\gamma_{n-1}}, \theta_{n-1}, \mathsf{tx}^{\mathsf{r}}_{n-1}) := \mathsf{left}(\texttt{pid})$
4. Extract $\theta_{\epsilon_{n-1}} \in \mathsf{tx}^{\mathsf{r}}_{n-1}.\mathsf{input}$
5. Extract T from $\theta_{n-1}[0].\phi$
6. Let $\alpha_i := \theta_{n-1}[0].\mathsf{cash}$. If not $0 \leq \alpha'_i \leq \alpha_i$, abort. Compute:
   - $\theta'_{n-1} := \texttt{genNewState}(\overline{\gamma_{n-1}}, \alpha'_{n-1}, T)$
   - $\mathsf{tx}^{\mathsf{r}'}_{n-1} := \texttt{genRefTx}(\theta'_{n-1}, \theta_{\epsilon_{n-1}}, U_{n-1})$
7. Create the signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{r}'}_{n-1})$
8. Send $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \theta'_{n-1}, \mathsf{tx}^{\mathsf{r}'}_{n-1}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{r}'}_{n-1})) \overset{\tau_0}{\hookrightarrow} U_{n-1}$.

Close: In every round, every node $U_i \in \mathcal{P}$ does the following. We denote τ_x as the current round.

U_i upon $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \theta'_i, \mathsf{tx}^{\mathsf{r}'}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \overset{\tau_x}{\hookleftarrow} U_{i+1}$

1. If $\texttt{pid} \notin \mathsf{pidSet}$, abort.
2. Retrieve $(\overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{r}}_i, \mathsf{sk}_{\widetilde{U_i}}) := \mathsf{right}(\texttt{pid})$
3. If $\overline{\gamma_i}.\mathsf{right} \neq U_{i+1}$, abort. If $\theta_i[0] \notin \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$, abort.
4. Extract $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
5. Extract T from $\theta_i[0].\phi$ and $\alpha_i := \theta_i[0].\mathsf{cash}$
6. Extract T' from $\theta'_i[0].\phi$ and $\alpha'_i := \theta'_i[0].\mathsf{cash}$
7. If $T' \neq T$, abort. If not $0 \leq \alpha'_i \leq \alpha_i$, abort.
8. If $\theta'_i \neq \texttt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$, abort.
9. If $\mathsf{tx}^{\mathsf{r}'}_i \neq \texttt{genRefTx}(\theta'_i, \theta_{\epsilon_i}, U_i)$, abort.

10. If $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}'})$ is not a valid signature for $\mathsf{tx}_i^{\mathsf{r}'}$, abort.
11. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE}, \alpha_i') \overset{\tau_x}{\hookrightarrow} \mathcal{E}$
12. If not $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSE\text{-}ACCEPT}) \overset{\tau_x}{\hookleftarrow} \mathcal{E}$, abort.
13. $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i') \overset{\tau_x}{\hookrightarrow} \mathcal{F}_{Channel}$.

U_{i+1} upon $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i') \overset{\tau_i+1+t_{\mathsf{u}}}{\hookleftarrow} \mathcal{F}_{Channel}$

14. Set $\mathsf{left}(\mathtt{pid}) := (\overline{\gamma_i}, \theta_i', \mathsf{tx}_i^{\mathsf{r}'})$

U_i upon $(\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i') \overset{\tau_x+t_{\mathsf{u}}}{\hookleftarrow} \mathcal{F}_{Channel}$

15. Let $\tau_i := \tau_x + t_{\mathsf{u}}$
16. Set $\mathsf{rightSig}(\mathtt{pid}) := \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}'})$ and set $\mathsf{right}(\mathtt{pid}) := (\overline{\gamma_i}, \theta_i', \mathsf{tx}_i^{\mathsf{r}'}, \mathsf{sk}_{\widetilde{U_i}})$.
17. If U_i is not the sending endpoint:
    - Retrieve $(\overline{\gamma_{i-1}}, \theta_{i-1}, \mathsf{tx}_{i-1}^{\mathsf{r}}) := \mathsf{left}(\mathtt{pid})$
    - Extract $\theta_{\epsilon_{i-1}} \in \mathsf{tx}_{i-1}^{\mathsf{r}}.\mathsf{input}$
    - $\theta_{i-1}' := \mathtt{genNewState}(\overline{\gamma_{i-1}}, \alpha_i' + \mathsf{fee}, T)$
    - $\mathsf{tx}_{i-1}^{\mathsf{r}'} := \mathtt{genRefTx}(\theta_{i-1}', \theta_{\epsilon_{i-1}}, U_{i-1})$
    - Create the signature $\sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})$
    - Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{close\text{-}req}, \theta_{i-1}', \mathsf{tx}_{i-1}^{\mathsf{r}'}, \sigma_{U_i}(\mathsf{tx}_{i-1}^{\mathsf{r}'})) \overset{\tau_i}{\hookrightarrow} U_{i-1}$.
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED}) \overset{\tau_i}{\hookrightarrow} \mathcal{E}$
18. If U_i is the sending endpoint:
    - $(\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED}) \overset{\tau_i}{\hookrightarrow} \mathcal{E}$

Emergency-Offload

U_0 in every round τ

For every entry $(\mathtt{pid}, \tau_{\mathsf{f}}, U_n) \in \mathsf{paySet}$ do the following:

1. Let $(\mathsf{tx}^{\mathsf{vc}}, T) := \mathsf{local}(\mathtt{pid})$.
2. If $\tau < T - t_{\mathsf{c}} - 3\Delta$, continue with next loop iteration.

3. Remove $(\texttt{pid}, \tau_{\mathsf{f}}, U_n)$ from paySet.
4. Sign $\mathsf{tx}^{\mathsf{vc}}$ yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})$ and set $\overline{\mathsf{tx}^{\mathsf{vc}}} := (\mathsf{tx}^{\mathsf{vc}}, (\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})))$. Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Respond

U_i at the end of every round

Let t be the current round. Do the following:

1. For every pid in $\mathsf{right}.\texttt{keyList}()$, let $(\overline{\gamma_i},\ \theta_i,\ \mathsf{tx}^{\mathsf{r}}_i,\ \mathsf{sk}_{\widetilde{U_i}}) := \mathsf{right}(\texttt{pid})$, let $(\mathsf{tx}^{\mathsf{vc}},\ T) := \mathsf{local}(\texttt{pid})$ and do the following. If $t < T - t_{\mathsf{c}} - 2\Delta$, $\mathsf{tx}^{\mathsf{vc}}$ is on the ledger $\mathcal{L}$ and $\overline{\gamma_i}.\mathsf{st} = \theta_i$, do the following:
   - Remove the entry for pid from right, send $(\texttt{ssid}_C, \texttt{CLOSE}, \overline{\gamma_i}.\mathsf{id}) \overset{t}{\hookrightarrow} \mathcal{F}_{Channel}$.
   - If a transaction tx with $\mathsf{tx}.\mathsf{output} = \theta_i$ is on $\mathcal{L}$ in round $t_1 \leq t + t_{\mathsf{c}}$ wait Δ rounds.
   - Sign $\mathsf{tx}^{\mathsf{r}}_i$ to yield $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i)$ and use $\mathsf{sk}_{\widetilde{U_i}}$ to sign $\mathsf{tx}^{\mathsf{r}}_i$ to yield $\sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)$
   - Set $\overline{\mathsf{tx}^{\mathsf{r}}_i} := (\mathsf{tx}^{\mathsf{r}}_i, (\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_i), \mathsf{rightSig}(\texttt{pid}), \sigma_{\widetilde{U_i}}(\mathsf{tx}^{\mathsf{r}}_i)))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{r}}_i}) \overset{t_1+\Delta}{\longrightarrow} \mathcal{G}_{Ledger}$. When it appears on $\mathcal{L}$ in round $t_2 < T$, send $(\texttt{sid}, \texttt{pid}, \texttt{REVOKED}) \overset{t_2}{\hookrightarrow} \mathcal{E}$
2. For every pid in $\mathsf{left}.\texttt{keyList}()$, let $(\overline{\gamma_{i-1}},\ \theta_{i-1},\ \mathsf{tx}^{\mathsf{r}}_{i-1}) := \mathsf{left}(\texttt{pid})$, let $(\mathsf{tx}^{\mathsf{vc}},\ T) := \mathsf{local}(\texttt{pid})$ and do the following. If $t \geq T$ and a transaction tx with $\mathsf{tx}.\mathsf{output} = \theta_{i-1}$ is on the ledger $\mathcal{L}$, but not $\mathsf{tx}^{\mathsf{r}}_{i-1}$, do the following:
   - Remove the entry for pid from left and create $\mathsf{tx}^{\mathsf{p}}_{i-1} := \texttt{genPayTx}(\overline{\gamma_{i-1}}.\mathsf{st}, U_i)$.
   - Sign $\mathsf{tx}^{\mathsf{p}}_{i-1}$ yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1})$.
   - Set $\overline{\mathsf{tx}^{\mathsf{p}}_{i-1}} := (\mathsf{tx}^{\mathsf{p}}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{p}}_{i-1}))$ and send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{p}}_{i-1}}) \overset{t}{\hookrightarrow} \mathcal{G}_{Ledger}$.
   - If it appears on $\mathcal{L}$ in round $t_1 \leq t + \Delta$, send $(\texttt{sid}, \texttt{pid}, \texttt{FORCE-PAY}) \overset{t_1}{\hookrightarrow} \mathcal{E}$

F.6.5 Simulation

In this section we provide the code for the simulator $\mathcal{S}$, which can simulate the protocol in the ideal world, and give the proof that the protocol (see Appendix F.6.4) UC-realizes the ideal functionality $\mathcal{F}_{Pay}$ shown in Appendix F.6.3.

Simulator
Local variables:

left	A map, storing the channel $\overline{\gamma_{i-1}}$ and output $\theta_{\epsilon_{i-1}}$ for a given keypair consisting of a payment id `pid` and a user U_i, or $(\bot, \bot)$ if U_i is the sending endpoint.
right	A map, storing the transaction $\mathsf{tx}_i^{\mathsf{r}}$ for a given keypair consisting of a payment id `pid` and a user U_i.
rightSig	A map, storing the signature of the right neighbor for the transaction stored in right for a given keypair consisting of a payment id `pid` and a user U_i.

Simulator for init phase

Upon $(\mathtt{sid}, \mathtt{init}) \xhookleftarrow{t_{\mathsf{init}}} \mathcal{F}_{Pay}$ and send $(\mathtt{sid}, \mathtt{init\text{-}ok}, t_{\mathsf{u}}, t_{\mathsf{c}}, t_{\mathsf{o}}) \xhookrightarrow{t_{\mathsf{init}}} \mathcal{F}_{Pay}$.

Simulator for Open-VC phase

Pre-create VC

1. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookleftarrow{\tau} U_0$ if U_0 dishonest, go to step (3).
2. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookleftarrow{\tau} \mathcal{F}_{Pay}$ if U_0 honest, do the following. If U_n honest go to step (3). If U_n dishonest, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T) \xhookrightarrow{\tau} U_n$ and go idle.
3. $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau) \xhookrightarrow{\tau} \mathcal{F}_{Channel}$.
4. If not $(\mathtt{ssid}_C, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id}) \xhookleftarrow{\tau+t_{\mathsf{o}}} \mathcal{F}_{Channel}$, mark VC as unusable.

a) Case U_i is honest, U_{i+1} dishonest

1. Upon receiving $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i}) \xhookleftarrow{\tau} \mathcal{F}_{Pay}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\mathtt{pid}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$.
2. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
3. $\theta_i := \mathtt{genStateOutputs}(\overline{\gamma_i}, \alpha_i, T)$
4. $\mathsf{tx}_i^{\mathsf{r}} := \mathtt{genRefTx}(\theta_i, \theta_{\epsilon_i}, U_i)$
5. $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}_i^{\mathsf{r}}) \xhookrightarrow{\tau} U_{i+1}$
6. Upon $(\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})) \xhookleftarrow{\tau+2} U_{i+1}$, check that $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ is a valid signature for $\mathsf{tx}_i^{\mathsf{r}}$. If not, go idle.
7. Set $\mathsf{rightSig}(\mathtt{pid}, U_i) := \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$, $\mathsf{right}(\mathtt{pid}, U_i) := \mathsf{tx}_i^{\mathsf{r}}$
8. Send $(\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xhookrightarrow{\tau+2} \mathcal{F}_{Channel}$.

9. If not $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i) \xleftarrow{\tau+2+t_u} \mathcal{F}_{Channel}$, go idle.
10. Set $\mathsf{left}(\texttt{pid}, U_i) := (\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}})$
11. $\text{Send}(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_i}, \theta_i, \mathsf{tx}^{\mathsf{vc}}, T, \theta_{\epsilon_i}, R) \hookrightarrow^{\tau} \mathcal{F}_{Pay}$.

b) Case U_i is honest, U_{i-1} dishonest

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{open-req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i, \theta_{i-1}, \mathsf{tx}^{\mathsf{r}}_{i-1}) \hookleftarrow^{\tau} U_{i-1}$.
   Let $\alpha_{i-1} := \theta_{i-1}[0].\mathsf{cash}$ and extract T from $\theta_{i-1}[0].\phi$ (the parameter of $\mathsf{AbsTime}()$). Let $\overline{\gamma_{i-1}}$ be the channel between U_{i-1} and U_i
2. Let $x := \texttt{checkTxEr}(U_i, U_i.a, U_i.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_i)$. Check that $x \neq \bot$, but instead $x = (\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1})$. Otherwise, go idle.
3. Check that there exists a channel between U_i and U_{i+1} and call this channel $\overline{\gamma_i}$. Verify that $\theta_{i-1} = \texttt{genStateOutputs}(\overline{\gamma_{i-1}}, \alpha_{i-1}, T)$ and $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\theta_{i-1}, \theta_{\epsilon_{i-1}}, U_i)$, where $\theta_{\epsilon_{i-1}} \in \mathsf{tx}^{\mathsf{vc}}$ and $\theta_{\epsilon_{i-1}} \neq \theta_{\epsilon_i}$.
4. $(\texttt{sid}, \texttt{pid}, \texttt{check-id}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i, U_{i-1}, U_i, U_{i+1}, \alpha_i, T) \hookrightarrow^{\tau} \mathcal{F}_{Pay}$
5. If not $(\texttt{sid}, \texttt{pid}, \texttt{ok}, \overline{\gamma_i}) \hookleftarrow^{\tau} \mathcal{F}_{Pay}$, go idle. Let $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
6. Sign $\mathsf{tx}^{\mathsf{r}}_{i-1}$ on behalf of U_i yielding $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})$ and $(\texttt{sid}, \texttt{pid}, \texttt{open-ok}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}}_{i-1})) \hookrightarrow^{\tau} U_{i-1}$.
7. Upon $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}, \theta_{i-1}) \xleftarrow{\tau+1+t_u} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{register}, \overline{\gamma_{i-1}}, \theta_{i-1}, \mathsf{tx}^{\mathsf{vc}}, T, \bot, \bot) \hookrightarrow^{\tau} \mathcal{F}_{Pay}$. Otherwise, go idle.
8. Set $\mathsf{left}(\texttt{pid}, U_i) := (\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}})$.
9. If $U_i = U_n$ (if $(\mathsf{sk}_{\widetilde{U_i}}, \theta_{\epsilon_i}, R_i, U_{i+1}, \mathsf{onion}_{i+1}) = (\top, \top, \top, \top, \top)$ holds), and U_0 is honest,[a] send $(\texttt{sid}, \texttt{pid}, \texttt{payment-open}, \mathsf{tx}^{\mathsf{vc}}) \xrightarrow{\tau+1+t_u} \mathcal{F}_{Pay}$. If U_0 is dishonest, create signature $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$ on behalf of U_n and send $(\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \xrightarrow{\tau+1+t_u} U_0$. In both cases, send via $\mathcal{F}_{Pay}$ to the dummy user U_n the message $(\texttt{sid}, \texttt{pid}, \texttt{PAYMENT-OPEN}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_{i-1}) \xrightarrow{\tau+1+t_u} U_n$. Go Idle.
10. Send via $\mathcal{F}_{Pay}$ to the dummy user U_i the message $(\texttt{sid}, \texttt{pid}, \texttt{OPENED}) \xrightarrow{\tau+1+t_u} U_i$.
11. If U_{i+1} honest, call $\text{process}(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{vc}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_i, T)$.
12. If U_{i+1} dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\texttt{pid}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_{i-1} - \mathsf{fee}, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$.

$\text{process}(\texttt{sid}, \texttt{pid}, \mathsf{tx}^{\mathsf{vc}}, \overline{\gamma_{i-1}}, \overline{\gamma_i}, R_i, \mathsf{onion}_i, \alpha_{i-1}, T)$

Let τ be the current round.

1. Initialize $\mathsf{nodeList} := \{U_i\}$ and $\mathsf{onions}, \mathsf{rMap}, \mathsf{stealthMap}$ as empty maps.

2. $(U_{i+1}, \mathsf{msg}_i, \mathsf{onion}_{i+1}) := \mathsf{GetRoutingInfo}(\mathsf{onion}_i)$
3. $\mathsf{stealthMap}(U_i) := \theta_{\epsilon_i}$
4. $\mathsf{rMap}(U_i) := R_i$
5. While U_i and U_{i+1} honest:
   - $x := \mathtt{checkTxEr}(U_{i+1}, U_{i+1}.a, U_{i+1}.b, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1})$:
     - If $x = \bot$, append U_{i+1} and then $\bot$ to $\mathsf{nodeList}$ and break the loop.
     - If $x = (\top, \top, \top, \top, \top)$, append U_{i+1} to $\mathsf{nodeList}$ and break the loop.
     - Else, if $x = (\mathsf{sk}_{\widetilde{U_{i+1}}}, \theta_{\epsilon_{i+1}}, U_{i+2}, \mathsf{onion}_{i+2})$, do the following.
   - Append U_{i+1} to $\mathsf{nodeList}$
   - $\mathsf{onions}(U_{i+2}) := \mathsf{onion}_{i+2}$
   - $\mathsf{rMap}(U_{i+1}) := R_{i+1}$
   - $\mathsf{stealthMap}(U_{i+1}) := \theta_{\epsilon_{i+1}}$
   - If U_{i+2} is dishonest, append U_{i+2} to $\mathsf{nodeList}$ and break the loop.
   - Set $i := i + 1$ (i.e., continue loop for U_{i+1} and U_{i+2})
6. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{continue}, \mathsf{nodeList}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{onions}, \mathsf{rMap}, \mathsf{rList}, \mathsf{stealthMap}, \alpha_{i-1}, T, \overline{\gamma_{i-1}}) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$

[a] For simplicity, assume that the U_n (and in the case it is honest, the simulator) knows the sender. As the payment is usually tied to the exchange of some goods, this is a reasonable assumption. Note that in practice, this is not necessary, as the sender can be embedded in the routing information onion_n.

Simulator for finalize and emergency-offload phase

a) Publishing $\mathsf{tx}^{\mathsf{vc}}$

Upon receiving a message $(\mathtt{sid}, \mathtt{pid}, \mathtt{denied}, \mathsf{tx}^{\mathsf{vc}}, U_0) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ and U_0 honest, sign $\mathsf{tx}^{\mathsf{vc}}$ on behalf of U_0 yielding $\sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}})$. Set $\overline{\mathsf{tx}^{\mathsf{vc}}} := (\mathsf{tx}^{\mathsf{vc}}, \sigma_{U_0}(\mathsf{tx}^{\mathsf{vc}}))$ and send $(\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

b) Case U_n honest, U_0 dishonest

Upon message $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$, sign $\mathsf{tx}^{\mathsf{vc}}$ on behalf of U_n yielding $\sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})$. Send $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \overset{\tau}{\hookrightarrow} U_0$.

c) Case U_n dishonest, U_0 honest

Upon message $(\mathtt{sid}, \mathtt{pid}, \mathtt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \overset{\tau}{\hookleftarrow} U_n$, send $(\mathtt{sid}, \mathtt{pid}, \mathtt{confirmed}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}})) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$.

Simulator for Close-VC phase

a) Case U_i is honest, U_{i-1} dishonest

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{close}, \alpha'_{i-1}, \overline{\gamma_{i-1}}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$ or upon being called by the simulator $\mathcal{S}$ itself in round τ with parameters $(\texttt{pid}, \alpha'_{i-1}, \overline{\gamma_{i-1}})$.
2. Retrieve $(\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}}) := \mathsf{left}(\texttt{pid}, U_i)$.
3. Extract T from $\overline{\gamma_{i-1}}.\mathsf{st}[0]$.
4. Let $U_i := \overline{\gamma_i}.\mathsf{left}$ and $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$.
5. $\theta'_{i-1} := \texttt{genNewState}(\overline{\gamma_{i-1}}, \alpha'_i, T)$
6. $\mathsf{tx}^{\mathsf{r}}_i := \texttt{genRefTx}(\theta'_{i-1}, \theta_{\epsilon_{i-1}}, U_{i-1})$
7. Create the signature $\sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})$ on U_i's behalf.
8. Send $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \theta'_{i-1}, \mathsf{tx}^{\mathsf{r}'}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1})) \overset{\tau}{\hookrightarrow} U_{i-1}$.
9. If $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_{i-1}}.\mathsf{id}, \theta'_{i-1}) \overset{\tau+1+t_{\mathsf{u}}}{\longleftarrow} \mathcal{F}_{Channel}$, send $(\texttt{sid}, \texttt{pid}, \texttt{replace}, \overline{\gamma_{i-1}}, \theta'_{i-1}) \overset{\tau+1+t_{\mathsf{u}}}{\longrightarrow} \mathcal{F}_{Pay}$.

b) Case U_i is honest, U_{i+1} dishonest

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{close-req}, \theta'_i, \mathsf{tx}^{\mathsf{r}'}_i, \sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)) \overset{\tau}{\hookleftarrow} U_{i+1}$, let $\overline{\gamma_i}$ the channel between U_i and U_{i+1}.
2. Let $\mathsf{tx}^{\mathsf{r}}_i := \mathsf{right}(\texttt{pid}, U_i)$. If no such entry exists, go idle.
3. Let $\theta_i := \overline{\gamma_i}.\mathsf{st}$ and check that $\theta_i[0] \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$. If not, go idle.
4. Extract $\theta_{\epsilon_i} \in \mathsf{tx}^{\mathsf{r}}_i.\mathsf{input}$
5. Extract T from $\theta_i[0].\phi$ and $\alpha_i := \theta_i[0].\mathsf{cash}$
6. Extract T' from $\theta'_i[0].\phi$ and $\alpha'_i := \theta'_i[0].\mathsf{cash}$
7. If $T' \neq T$, abort. If not $0 \leq \alpha'_i \leq \alpha_i$, abort.
8. If $\theta'_i \neq \texttt{genNewState}(\overline{\gamma_i}, \alpha'_i, T)$, abort.
9. If $\mathsf{tx}^{\mathsf{r}'}_i \neq \texttt{genRefTx}(\theta'_i, \theta_{\epsilon_i}, U_i)$, abort.
10. If $\sigma_{U_{i+1}}(\mathsf{tx}^{\mathsf{r}'}_i)$ is not a valid signature for $\mathsf{tx}^{\mathsf{r}'}_i$, abort.
11. Via $\mathcal{F}_{Pay}$ to the dummy user U_i send $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i) \overset{\tau}{\hookrightarrow} U_i$ and expect the answer $(\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT}) \overset{\tau}{\hookleftarrow} U_i$, otherwise go idle.
12. Send $(\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta'_i) \overset{\tau}{\hookrightarrow} \mathcal{F}_{Channel}$.
13. Expect $(\texttt{ssid}_C, \texttt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta'_i) \overset{\tau+t_{\mathsf{u}}}{\longleftarrow} \mathcal{F}_{Channel}$, else go idle.

14. Send $(\texttt{sid}, \texttt{pid}, \texttt{replace}, \overline{\gamma_i}, \theta_i') \overset{\tau}{\hookrightarrow} \mathcal{F}_{Pay}$.
15. Set $\mathsf{right}(\texttt{pid}, U_i) := \mathsf{tx}_i^{\mathsf{r}'}$
16. Retrieve $(\overline{\gamma_{i-1}}, \theta_{\epsilon_{i-1}}) := \mathsf{left}(\texttt{pid}, U_i)$.
17. If $U_i = U_0$, send via $\mathcal{F}_{Pay}$ to the dummy user U_i the message $(\texttt{sid}, \texttt{pid}, \texttt{VC-CLOSED}) \overset{\tau+t_u}{\hookrightarrow} U_i$. Go idle.
18. Send via $\mathcal{F}_{Pay}$ to the dummy user U_i the message $(\texttt{sid}, \texttt{pid}, \texttt{CLOSED}) \overset{\tau+t_u}{\hookrightarrow} U_i$.
19. If U_{i-1} honest, send $(\texttt{sid}, \texttt{pid}, \texttt{continue-close}, \overline{\gamma_{i-1}}, \alpha_i' + \mathsf{fee}) \overset{\tau+t_u}{\hookrightarrow} \mathcal{F}_{Pay}$
20. If dishonest, go to step Simulator U_i honest, U_{i+1} dishonest step 1 with parameters $(\texttt{pid}, \alpha_i' + \mathsf{fee}, \overline{\gamma_{i-1}})$.

Simulator for respond phase

In every round τ, upon receiving the following two messages, react accordingly.

1. Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-refund}, \overline{\gamma_i}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_i}, R_i) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$.
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$.
   - If U_{i+1} is honest, create the transaction $\mathsf{tx}_i^{\mathsf{r}} := \texttt{genRefTx}(\overline{\gamma_i}.\mathsf{st}[0],\ \theta_{\epsilon_i},\ U_i)$. Else, let $\mathsf{tx}_i^{\mathsf{r}} := \mathsf{right}(\texttt{pid}, U_i)$
   - Extract $\mathsf{pk}_{\widetilde{U_i}}$ from output θ_{ϵ_i} of $\mathsf{tx}^{\mathsf{vc}}$ and let $\mathsf{sk}_{\widetilde{U_i}} := \mathsf{GenSk}(U_i.a, U_i.b, \mathsf{pk}_{\widetilde{U_i}}, R_i)$.
   - Generate signatures $\sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}})$ and, using $\mathsf{sk}_{\widetilde{U_i}}$, $\sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})$ on behalf of U_i.
   - If $U_{i+1} := \overline{\gamma_i}.\mathsf{right}$ is honest, generate signature $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}})$ on behalf of U_{i+1}. Else, let $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}) := \mathsf{rightSig}(\texttt{pid}, U_i)$
   - Set $\overline{\mathsf{tx}_i^{\mathsf{r}}} := (\mathsf{tx}_i^{\mathsf{r}}, (\sigma_{U_i}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}), \sigma_{\widetilde{U_i}}(\mathsf{tx}_i^{\mathsf{r}})))$.
   - Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.
2. Upon $(\texttt{sid}, \texttt{pid}, \texttt{post-pay}, \overline{\gamma_i}) \overset{\tau}{\hookleftarrow} \mathcal{F}_{Pay}$
   - Extract α_i and T from $\overline{\gamma_i}.\mathsf{st}.\mathsf{output}[0]$. Create the transaction $\mathsf{tx}_i^{\mathsf{p}} := \texttt{genPayTx}(\overline{\gamma_i}.\mathsf{st}, U_{i+1})$.
   - Generate signatures $\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{p}})$ and set $\overline{\mathsf{tx}_i^{\mathsf{p}}} := (\mathsf{tx}_i^{\mathsf{p}}, (\sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{p}})))$.
   - Send $(\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}_i^{\mathsf{p}}}) \overset{\tau}{\hookrightarrow} \mathcal{G}_{Ledger}$.

Proof.

We proceed to show that for any environment $\mathcal{E}$ an interaction with $\phi_{\mathcal{F}_{Pay}}$ (the ideal protocol of ideal functionality $\mathcal{F}_{Pay}$) via the dummy parties and $\mathcal{S}$ (ideal world) is indistinguishable from an interaction with Π and an adversary $\mathcal{A}$. More formally, we show that the execution ensembles $\texttt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}}$ and $\texttt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ are indistinguishable for the environment $\mathcal{E}$.

We use the notation $m[\tau]$ to denote that a message m is observed by $\mathcal{E}$ at round τ. We interact with other ideal functionalities. These functionalities might in turn interact with the environment or parties under adversarial control, either by sending messages or by impacting public variables, i.e., the ledger $\mathcal{L}$. To capture this impact, we define a function $\mathsf{obsSet}(m, \mathcal{F}, \tau)$, returning a set of all by $\mathcal{E}$ observable actions which are triggered by calling $\mathcal{F}$ with message m in round τ.

In this proof, we do a case-by-case analysis of each corruption setting. We start with the view of the environment in the real world and follow with the view in the ideal world, simulated by $\mathcal{S}$. Due to the similarities of the Open-VC, the Finalize well as the Respond phase and the Pay, Finalize and Respond phase in [AMSKM21], parts of the corresponding proofs are taken verbatim from there.

Lemma 26. *Let Σ be an EUF-CMA secure signature scheme. Then, the Open-VC phase of Π GUC-emulates the Open-VC phase of functionality $\mathcal{F}_{Pay}$.*

Proof. We compare the execution ensembles for the open phase in the real and the ideal world. In Table F.3 we match the sequence of the Open-VC phase of the ideal and the real world and point to which code is executed. We divide this phase into setup and open. For readability, we define the following messages:

- $m_0 := (\mathtt{sid}, \mathtt{pid}, \mathtt{pre\text{-}create\text{-}vc}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, T)$
- $m_1 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PRE\text{-}CREATE}, \overline{\gamma_{\mathsf{vc}}}, \mathsf{tx}^{\mathsf{vc}}, 0, T - \tau)$
- $m_2 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PRE\text{-}CREATED}, \overline{\gamma_{\mathsf{vc}}}.\mathsf{id})$
- $m_3 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}req}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \theta_i, \mathsf{tx}_i^{\mathsf{r}})$
- $m_4 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPEN}, \mathsf{tx}^{\mathsf{vc}}, \theta_{\epsilon_{i+1}}, R_i, U_i, U_{i+2}, \alpha_i, T)$
- $m_5 := (\mathtt{sid}, \mathtt{pid}, \mathtt{ACCEPT}, \overline{\gamma_{i+1}})$
- $m_6 := (\mathtt{sid}, \mathtt{pid}, \mathtt{open\text{-}ok}, \sigma_{U_{i+1}}(\mathsf{tx}_i^{\mathsf{r}}))$
- $m_7 := (\mathtt{ssid}_C, \mathtt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta_i)$
- $m_8 := (\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i)$
- $m_9 := (\mathtt{sid}, \mathtt{pid}, \mathtt{OPENED})$ or, if sent by the receiver,
  $m_9 := (\mathtt{sid}, \mathtt{pid}, \mathtt{PAYMENT\text{-}OPEN}, \mathsf{tx}^{\mathsf{vc}}, T, \alpha_i)$

Setup.

Table F.3: Explanation of the sequence names used in Lemma 26 and where they can be found in the ideal functionality (IF), Protocol (Prot) or Simulator (Sim).

	Real World	Ideal World			Output	Description
		U_i honest, U_{i+1} corrupted	U_i honest, U_{i+1} honest	U_i corrupted, U_{i+1} honest		
SETUP	Prot.OpenVC.Setup 1-15	IF.OpenVC.Setup 1-6, Sim.OpenVC.PrecreateVC 1-4, IF.OpenVC.Setup 7-10,12, Sim.OpenVC.a 1-5	IF.OpenVC.Setup 1-6, Sim.OpenVC.PrecreateVC 1-4, IF.OpenVC.Setup 7-11	Sim.OpenVC.PrecreateVC 1-4	m_0, $2 \cdot m_1$, m_3	Pre-Creates VC, performs setup and contacts next user
CREATE_STATE	Prot.OpenVC.Open 6-8	IF.OpenVC.Open 12,13 , Sim.OpenVC.a 1-5	IF.OpenVC.Open 12, 13	Sim.OpenVC.b 8-12	m_9, m_3	Upon m_8, sends message m_9 to $\mathcal{E}$. Then, ceates the objects to send in m_3 and sends it to next user (or finalize).
CHECK_STATE	Prot.OpenVC.Open 1-4	n/a	IF.OpenVC.Open 1-8	Sim.OpenVC.b 1-4 IF.Check Sim.OpenVC.b 5-7 IF.Register	m_4, m_6	Checks if objects in m_3 are correct, sends m_4 to $\mathcal{E}$ and on m_5, sends m_6 to U_i
CHECK_SIG	Prot.OpenVC.Open 5	Sim.OpenVC.a 6-11	IF.OpenVC.Open 9-11	n/a	m_7	Checks if signature of tx'_i is correct

Real world: An honest U_0 performs SETUP in τ_0 to set up the initial objects and to pre-create the VC with U_n. In round τ_0, U_0 sends m_0 to U_n (which $\mathcal{E}$ sees in round $\tau_0 + 1$ only if U_n is corrupted) and then, after waiting 1 round, m_1 to $\mathcal{F}_{Channel}$. Note that an honest U_n receiving m_0 in some round τ, sends also a message m_1 to $\mathcal{F}_{Channel}$. If $\mathcal{F}_{Channel}$ received two valid messages m_1 from U_0 and U_n, it returns m_2. Depending on the corruption setting, the ensemble

- $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_0[\tau_0 + 1]\} \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n corrupted
- $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1) \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n honest, where m_1 is sent by each user.
- $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau)$ for U_0 corrupted, U_n honest

Ideal world: For an honest U_0, $\mathcal{F}_{Pay}$ performs SETUP in τ_0 to set up the initial objects and to pre-create the VC. In round τ_0, $\mathcal{F}_{Pay}$ asks $\mathcal{S}$ to send m_0 to a dishonest U_n (who receives it in round $\tau_0 + 1$), or, if U_n is honest send m_1 to $\mathcal{F}_{Channel}$ in $\tau_0 + 1$ on behalf of U_n. In both cases, $\mathcal{F}_{Pay}$ sends m_1 to $\mathcal{F}_{Channel}$ in $\tau_0 + 1$. If U_0 is dishonest and U_n honest, $\mathcal{S}$ waits for a message m_0 from U_0 in some round τ and sends m_1 to $\mathcal{F}_{Channel}$. If $\mathcal{F}_{Channel}$ received two valid messages m_1 from U_0 and U_n, it returns m_2. Depending on the corruption setting, the ensemble

- $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_0[\tau_0 + 1]\} \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n corrupted
- $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1) \cup \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau_0 + 1)$ for U_0 honest, U_n honest, where m_1 is sent for each user.
- $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_1, \mathcal{F}_{Channel}, \tau)$ for U_0 corrupted, U_n honest

Open. 1. U_i honest, U_{i+1} corrupted.

Real world: After U_i performs either SETUP or CREATE_STATE, it sends m_3 to U_{i+1} in the current round τ. The environment $\mathcal{E}$ controls $\mathcal{A}$ and therefore U_{i+1} and will see m_3 in round $\tau + 1$. Iff U_{i+1} replies with a correct message m_6 in $\tau + 2$, U_i will

perform CHECK_SIG and call $\mathcal{F}_{Channel}$ with message m_7 in the same round. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_3[\tau+1]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau+2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either SETUP or simulator performs CREATE_STATE, the simulator sends m_3 to U_{i+1} in the current round τ. $\mathcal{E}$ will see m_3 in round $\tau+1$. Iff U_{i+1} replies with a correct message m_6 in $\tau+2$, the simulator will perform CHECK_SIG and call $\mathcal{F}_{Channel}$ with message m_7 in the same round. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_3[\tau+1]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau+2)$

2. U_i **honest,** U_{i+1} **honest.**

Real world: After U_i performs either SETUP or CREATE_STATE, it sends m_3 to U_{i+1} in the current round τ. U_{i+1} performs CHECK_STATE and sends m_4 to $\mathcal{E}$ in round $\tau+1$. Iff $\mathcal{E}$ replies with m_5, U_{i+1}, U_{i+1} replies with m_6. U_i receives this in round $\tau+2$, performs CHECK_SIG and sends m_7 to $\mathcal{F}_{Channel}$. U_{i+1} expects the message m_8 in round $\tau+2+t_{\mathsf{u}}$ and will then send m_9 to $\mathcal{E}$. Afterwards it continues with either CREATE_STATE or FINALIZE. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_4[\tau+1], m_9[\tau+2+t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau+2)$

Ideal world: After $\mathcal{F}_{Pay}$ performs either SETUP or is invoked by itself (in step Open.13) or by the simulator (in step process.6) in the current round τ, $\mathcal{F}_{Pay}$ perform the procedure Open. This behaves exactly like CREATE_STATE, CHECK_STATE and CHECK_SIG. However, since every object is created by $\mathcal{F}_{Pay}$, the checks are omitted. The procedure Open outputs the messages m_4 in round $\tau+1$ and iff $\mathcal{E}$ replies with m_5, calls $\mathcal{F}_{Channel}$ with m_7 in $\tau+2$. Finally, if m_8 is received in round $\tau+2+t_{\mathsf{u}}$, outputs m_9 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_4[\tau+1], m_9[\tau+2+t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_7, \mathcal{F}_{Channel}, \tau+2)$

3. U_i **corrupted,** U_{i+1} **honest.**

Real world: After U_{i+1} receives the message m_3 from U_i, it performs CHECK_STATE and sends m_4 to $\mathcal{E}$ in the current round τ. Iff $\mathcal{E}$ replies with m_5, U_{i+1} sends m_6 to U_i. If U_{i+1} receives the message m_8 from $\mathcal{F}_{Channel}$ in round $\tau+1+t_{\mathsf{u}}$, it sends m_9 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_4[\tau], m_6[\tau+1], m_9[\tau+1+t_{\mathsf{u}}]\}$

Ideal world: After the simulator receives m_3 from U_i, it performs CHECK_STATE together with $\mathcal{F}_{Pay}$ and $\mathcal{F}_{Pay}$ sends m_4 to $\mathcal{E}$. Iff $\mathcal{E}$ replies with m_5, $\mathcal{F}_{Pay}$ asks the simulator to send m_6 to U_i. All of this happens in the current round τ. If the simulator receives m_8 in round $\tau+1+t_{\mathsf{u}}$, it sends m_9 to $\mathcal{E}$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_4[\tau], m_6[\tau+1], m_9[\tau+1+t_{\mathsf{u}}]\}$

Note that we do not care about the case where both U_i and U_{i+1} are corrupted because the environment is communicating with itself, which is trivially the same in the ideal and

the real world. We see that for the setup and open phase in all three corruption cases, the execution ensembles of the ideal and the real world are identical, thereby proving Lemma 26.

□

Lemma 27. *Let Σ be a EUF-CMA secure signature scheme. Then, the Finalize phase of protocol Π GUC-emulates the Finalize phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles of the interaction between users U_n and U_0 for three different cases. We match the sequences and where they are used in the ideal and real world in Table F.4. We define the following messages.

- $m_{10} := (\texttt{sid}, \texttt{pid}, \texttt{finalize}, \mathsf{tx}^{\mathsf{vc}}, \sigma_{U_n}(\mathsf{tx}^{\mathsf{vc}}))$
- $m_{11} := (\texttt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}})$

Table F.4: Explanation of the sequence names used in Lemma 27 and where they can be found.

	Real World	Ideal World			Output	Description
		U_n honest, U_0 corrupted	U_n honest, U_0 honest	U_n corrupted, U_0 honest		
FINALIZE	Prot.OpenVC.Open 8	IF.OpenVC.12 and Sim.Finalize.b or Sim.OpenVC.b 9	IF.OpenVC.12 or Sim.OpenVC.b 9, IF.VCOpen	n/a	m_{10}	Sends finalize message to U_0
CHECK_FINALIZE	Prot.Finalize 1-3	n/a	IF.Finalize 1,2,4 Sim.Finalize.a	Sim.Finalize.c IF.Finalize 1,3,4 Sim.Finalize.a	m_{11}	Checks if $\mathsf{tx}^{\mathsf{vc}}$ is the same, if not, publishes it to ledger with m_{11}.

1. U_n honest, U_0 corrupted.

Real world: After performing FINALIZE in the current round τ, U_n sends m_{10} to U_0, which $\mathcal{E}$ sees in $\tau + 1$. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{10}[\tau + 1]\}$

Ideal world: After either $\mathcal{F}_{Pay}$ or the simulator performs FINALIZE in the current round τ, the simulator sends m_{10} to U_0, which $\mathcal{E}$ sees in $\tau + 1$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{10}[\tau + 1]\}$

2. U_n honest, U_0 honest.

Real world: After performing FINALIZE in the current round τ, U_n sends m_{10} to U_0. In the meantime, user U_0 performs CHECK_FINALIZE and should it not receive a correct message m_{10} in the correct round, will send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

Ideal world: Either $\mathcal{F}_{Pay}$ or the simulator performs FINALIZE in the current round τ. In the meantime, functionality $\mathcal{F}_{Pay}$ performs CHECK_FINALIZE and will, if the checks in FINALIZE failed or it was performed in a incorrect round τ', $\mathcal{F}_{Pay}$ will instruct the simulator to send m_{11} to $\mathcal{G}_{Ledger}$ in rounds τ'. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

3. U_n corrupted, U_0 honest.

Real world: U_0 performs CHECK_FINALIZE and should it not receive a correct message m_{10} in the correct round, will send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

Ideal world: The simulator and $\mathcal{F}_{Pay}$ perform CHECK_FINALIZE and should the simulator not receive a correct message m_{10} in the correct round, $\mathcal{F}_{Pay}$ will instruct the simulator to send m_{11} to $\mathcal{G}_{Ledger}$ in round τ'. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau')$

□

Lemma 28. *Let Σ be a EUF-CMA secure signature scheme. Then, the Update phase of protocol Π GUC-emulates the Update phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Trivially, this update phase is the same, as the pre-update messages are simply forwarded to $\mathcal{F}_{Channel}$ in both the real and the ideal world. □

Lemma 29. *Let Σ be a EUF-CMA secure signature scheme. Then, the Close phase of protocol Π GUC-emulates the Close phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles of the interaction between users U_{i+1} and U_i for three different cases. We match the sequences and where they are used in the ideal and real world in Table F.5. We define the following messages.

- $m_{12} := (\texttt{sid}, \texttt{pid}, \texttt{close-req}, \theta'_{i-1}, \mathsf{tx}^{\mathsf{r}'}_{i-1}, \sigma_{U_i}(\mathsf{tx}^{\mathsf{r}'}_{i-1}))$
- $m_{13} := (\texttt{sid}, \texttt{pid}, \texttt{CLOSE}, \alpha'_i)$
- $m_{14} := (\texttt{sid}, \texttt{pid}, \texttt{CLOSE-ACCEPT})$
- $m_{15} := (\texttt{ssid}_C, \texttt{UPDATE}, \overline{\gamma_i}.\mathsf{id}, \theta'_i)$

Table F.5: Explanation of the sequence names used in Lemma 29 and where they can be found.

	Real World	Ideal World			Output	Description
		U_{i+1} honest, U_i corrupted	U_{i+1} honest, U_i honest	U_{i+1} corrupted, U_i honest		
SHUTDOWN	Prot.CloseVC.Shutdown 1-8	IF.CloseVC.Start 1-3,5, Sim.CloseVC.a 1-8	IF.CloseVC.Start 1-4	n/a	m_{12}	Shutdown starts with U_n, creates objects, contacts next user
CLOSE	Prot.CloseVC.Close 1-14	n/a	IF.CloseVC.Close 1-10	Sim.CloseVC.b 1-12	m_{13}, m_{15}	Checks if objects in m_{12} are correct, sends m_{13} to $\mathcal{E}$ and on m_{14}, sends m_{15} to $\mathcal{F}_{Channel}$
PROCEED_CLOSE	Prot.CloseVC.Close 15-18	IF.CloseVC.Close 11,12, Sim.CloseVC.a	IF.CloseVC.Close 11,12	Sim.CloseVC.b 13,14, IF.CloseVC.Replace, Sim.CloseVC.B 14-20	m_{17}	On m_{16}, sends m_{17} to $\mathcal{E}$ and continues with next user (if exists).

- $m_{16} := (\mathtt{ssid}_C, \mathtt{UPDATED}, \overline{\gamma_i}.\mathsf{id}, \theta_i')$
- $m_{17} := (\mathtt{sid}, \mathtt{pid}, \mathtt{CLOSED})$ or, if sent by the sender, $m_{17} := (\mathtt{sid}, \mathtt{pid}, \mathtt{VC\text{-}CLOSED})$

1. U_{i+1} honest, U_i corrupted.

Real world: After U_{i+1} performs either SHUTDOWN or alternatively PROCEED_CLOSE, it sends m_{12} to U_i in the current round τ. The environment $\mathcal{E}$ controls $\mathcal{A}$ and therefore U_i and will see m_{12} in round $\tau + 1$. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{12}[\tau + 1]\}$

Ideal world: After $\mathcal{F}_{Pay}$ performs either SHUTDOWN or $\mathcal{S}$ performs PROCEED_CLOSE, the simulator sends m_{12} to U_i in the current round τ. $\mathcal{E}$ will see m_{12} in round $\tau+1$. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{12}[\tau + 1]\}$

2. U_{i+1} honest, U_i honest.

Real world: After U_{i+1} performs either SHUTDOWN or alternatively PROCEED_CLOSE, it sends m_{12} to U_i in the current round τ. U_i receives this message in $\tau + 1$ and carries out CLOSE, sending m_{13} to $\mathcal{E}$ in $\tau + 1$ and, upon m_{14} in $\tau + 1$, sends m_{15} in $\tau + 1$ to $\mathcal{F}_{Channel}$. After a successful update (m_{16} is received), U_i sends m_{17} to $\mathcal{E}$ in $\tau + 1 + t_{\mathsf{u}}$ and continues with U_{i-1}, if it exists. The ensemble is $\mathtt{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau + 1], m_{17}[\tau + 1 + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau + 1, \mathcal{F}_{Channel})$.

Ideal world: After $\mathcal{F}_{Pay}$ performs either SHUTDOWN or is invoked by itself (in step Close.12) or by the simulator (in step b.19 and then IF.Continue-Close) in the current round τ, $\mathcal{F}_{Pay}$ perform the procedure Close. This behaves exactly like CLOSE and PROCEED_CLOSE. However, since every object is created by $\mathcal{F}_{Pay}$, the checks are omitted. The procedure Close outputs the messages m_{13} in round $\tau + 1$ and iff $\mathcal{E}$ replies with m_{14}, calls $\mathcal{F}_{Channel}$ with m_{15} in $\tau+1$. Finally, if m_{16} is received in round $\tau+1+t_{\mathsf{u}}$, outputs m_{17} to $\mathcal{E}$ and continues for U_{i-1}, if it exists. The ensemble is $\mathtt{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau + 1], m_{17}[\tau + 1 + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau + 1, \mathcal{F}_{Channel})$

3. U_{i+1} corrupted, U_i honest.

Real world: After U_i receives the message m_{12} from U_{i+1} in round τ, it performs CLOSE and sends m_{13} to $\mathcal{E}$ in τ. Iff $\mathcal{E}$ replies with m_{14} in the same round, U_i sends m_{15} to $\mathcal{F}_{Channel}$ in τ. After receiving m_{16} in $\tau + t_{\mathsf{u}}$, performs PROCEED_CLOSE, sending m_{17} to $\mathcal{E}$ and continues with U_{i-1}, if it exists. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

Ideal world: After the $\mathcal{S}$ receives m_{12} from U_{i+1} in round τ, performs the steps CLOSE, sending m_{13} to $\mathcal{E}$ in τ. Iff $\mathcal{E}$ replies with m_{14} in the same round, $\mathcal{S}$ sends m_{15} to $\mathcal{F}_{Channel}$ in τ. After receiving m_{16} in $\tau + t_{\mathsf{u}}$, $\mathcal{S}$ performs PROCEED_CLOSE together with $\mathcal{F}_{Pay}$, sending m_{17} to $\mathcal{E}$ and continues for U_{i-1}, if it exists. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{13}[\tau], m_{17}[\tau + t_{\mathsf{u}}]\} \cup \mathsf{obsSet}(m_{15}, \tau, \mathcal{F}_{Channel})$.

□

Lemma 30. *Let Σ be a EUF-CMA secure signature scheme. Then, the Emergency-Offload phase of protocol Π GUC-emulates the Emergency-Offload phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles, but now only for an honest U_0. We use message $m_{11} := (\mathtt{ssid}_L, \texttt{POST}, \overline{\mathsf{tx}^{\mathsf{vc}}})$ from before.

Real world: An honest U_0 checks every round and each of its VCs (with a certain pid), if the VC has already been closed, see Prot.EmergencyOffload 1-4. If it has not within a certain round τ, U_0 sends m_{11} to $\mathcal{G}_{Ledger}$ in τ. The ensemble is $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

Ideal world: $\mathcal{F}_{Pay}$ checks every round and every VC (with a certain pid), if the VC has already been closed. If it has not within a certain round τ, $\mathcal{F}_{Pay}$ instructs $\mathcal{S}$ to send m_{11} to $\mathcal{G}_{Ledger}$, see IF.EmergencyOffload 1-4 and Sim.Finalize.a. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \mathsf{obsSet}(m_{11}, \mathcal{G}_{Ledger}, \tau)$.

□

Lemma 31. *Let Σ be a EUF-CMA secure signature scheme. Then, the Respond phase of protocol Π GUC-emulates the Respond phase of functionality $\mathcal{F}_{Pay}$.*

Proof. Again, we consider the execution ensembles. This time only for the case where a user U_i is honest, however, we distinguish between the case of revoke and force-pay. We match the sequences and where they are used in the ideal and real world in Table F.6. We define the following messages.

- $m_{18} := (\mathtt{ssid}_C, \mathtt{CLOSE}, \overline{\gamma_i.\mathsf{id}})$
- $m_{19} := (\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_i^{\mathsf{r}}})$
- $m_{20} := (\mathtt{sid}, \mathtt{pid}, \mathtt{REVOKED})$
- $m_{21} := (\mathtt{ssid}_L, \mathtt{POST}, \overline{\mathsf{tx}_{i-1}^{\mathsf{p}}})$
- $m_{22} := (\mathtt{sid}, \mathtt{pid}, \mathtt{FORCE\text{-}PAY})$

Table F.6: Explanation of the sequence names used in Lemma 31 and where they can be found.

	Real World	Ideal World	Output	Description
		U_i honest		
RESPOND	Prot.Respond	IF.Respond	n/a	Checks every round if response in order.
REVOKE	Prot.Respond.1	IF.Respond.Revoke Sim.Respond.1	m_{18}, m_{19}, m_{20}	Carries out the revokation.
FORCE_PAY	Prot.Respond.2	IF.Respond.Revoke Sim.Respond.2	m_{21}, m_{22}	Carries out the force-pay.

U_i honest, revoke.

Real world: In every round τ, U_i performs RESPOND, which provides a decision on whether or not to do the following. If yes, U_i performs REVOKE, which results in message m_{18} to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_{18} is closed, U_i sends m_{19} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{19} appears on $\mathcal{L}$ in $\tau + t_{\mathsf{c}} + 2\Delta$, U_i sends m_{20} to $\mathcal{E}$. The ensemble is $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{20}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_{18}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{19}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs RESPOND, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform REVOKE, which results in the message m_{18} to $\mathcal{F}_{Channel}$ in round τ. If the channel that is sent in m_{18} is closed, the simulator sends m_{19} to $\mathcal{G}_{Ledger}$ in round $\tau + t_{\mathsf{c}} + \Delta$. Finally, if the transaction sent in m_{19} appears on $\mathcal{L}$, $\mathcal{F}_{Pay}$ sends m_{20} to $\mathcal{E}$. The ensemble is $\mathrm{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{20}[\tau + t_{\mathsf{c}} + 2\Delta]\} \cup \mathsf{obsSet}(m_{18}, \mathcal{F}_{Channel}, \tau) \cup \mathsf{obsSet}(m_{19}, \mathcal{G}_{Ledger}, \tau + t_{\mathsf{c}} + \Delta)$

U_i honest, force-pay.

Real world: In every round τ, U_i performs RESPOND, which provides a decision on whether or not to do the following. If yes, U_i performs FORCE_PAY, which results in the messages m_{21} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{21} appears on $\mathcal{L}$, the message m_{22} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\mathrm{EXEC}_{\Pi,\mathcal{A},\mathcal{E}} := \{m_{22}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

Ideal world: In every round τ, $\mathcal{F}_{Pay}$ performs RESPOND, which provides a decision on whether or not to do the following. If yes, $\mathcal{F}_{Pay}$ instructs the simulator to perform FORCE_PAY, which results in the messages m_{21} to $\mathcal{G}_{Ledger}$ in round τ and, if the transaction sent in m_{21} appears on $\mathcal{L}$, the message m_{22} to $\mathcal{E}$ in round $\tau + \Delta$. The ensemble is $\text{EXEC}_{\mathcal{F}_{Pay},\mathcal{S},\mathcal{E}} := \{m_{22}[\tau + \Delta]\} \cup \mathsf{obsSet}(m_{21}, \mathcal{G}_{Ledger}, \tau)$

□

Theorem 7 (restated)**.** *Let Σ be an EUF-CMA secure signature scheme. Then, for functionalities $\mathcal{G}_{Ledger}$, $\mathcal{G}_{clock}$, $\mathcal{F}_{GDC}$, $\mathcal{F}_{Channel}$ and for any ledger delay $\Delta \in \mathbb{N}$, the protocol Π UC-realizes the ideal functionality $\mathcal{F}_{Pay}$.*

This theorem follows directly from Lemma 26, 27, 28, 29, 30 and Lemma 31.

F.6.6 Discussion on security and privacy goals

We state our security and privacy goals informally in Section 7.5.1. In this section, we formally define these goals as cryptographic games on top of the ideal functionality $\mathcal{F}_{Pay}$ described in Appendix F.6.3 and then show that $\mathcal{F}_{Pay}$ fulfills each goal. Due to the same assumptions and similarities in some of the security and privacy goals, parts of this section are taken verbatim from [AMSKM21].

Assumptions

For the theorems in this section, we have the following assumptions: (i) stealth addresses achieve unlinkability and (ii) the used routing scheme (i.e., Sphinx extended with a per-hop payload) is a secure onion routing process.

Unlinkability of stealth addresses. Consider the following game. The challenger computes two pair of stealth addresses (A_0, B_0) and (A_1, B_1). Moreover, the challenger picks a bit b and computes $P_b, R_b \leftarrow \mathsf{GenPk}(A_b, B_b)$. Finally, the challenger sends the tuples (A_0, B_0), (A_1, B_1) and P_b, R_b to the adversary.

Additionally, the adversary has access to an oracle that upon being queried, it returns P_b^*, R_b^* to the adversary.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 20 (Unlinkability of Stealth Addresses)**.** We say that a stealth addresses scheme achieves unlinkability if for all PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Secure onion routing process. We say that an onion routing process is secure, if it realizes the ideal functionality defined in [CL05]. Sphinx [DG09], for instance, is a realization of this. We use it in Donner, extended with a per-hop payload (see also Section 7.5.2).

Balance security

Given a path $\mathsf{channelList} := \gamma_0, \ldots, \gamma_{n-1}$ and given a user U such that $\gamma_i.\mathsf{right} = U$ and $\gamma_{i+1}.\mathsf{left} = U$, we say that the balance of U in the path is $\mathbf{PathBalance}(U) := \gamma_i.\mathsf{balance}(U) + \gamma_{i+1}.\mathsf{balance}(U)$. Intuitively then, we say that a virtual channel (VC) protocol achieves *balance security* if the $\mathbf{PathBalance}(U)$ for each honest intermediary U does not decrease.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input (`sid`, `pid`, `SETUP`, $\mathsf{channelList}$, $\mathsf{tx}^{\mathsf{in}}$, α, T, $\overline{\gamma_0}$). Every time a corrupted user U_i needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer, thereby giving the attacker the opportunity to stop opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC, or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

We say that the adversary wins the game if there exists an honest intermediate user U, such that $\mathbf{PathBalance}(U)$ is lower after the VC execution.

Definition 21 (Balance security)**.** We say that a VC protocol achieves balance security if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 19 (Donner achieves balance security)**.** *Donner virtual channel executions achieve balance security as defined in Definition 21.*

Proof. Assume that an adversary exists, and can win the balance security game. This means, that after the balance security game, there exists an honest intermediate user U, such $\mathbf{PathBalance}(U)$ is lower after the VC execution.

An intermediary U_i potentially has coins locked up in the state stored in $\mathcal{F}_{Channel}$ with its left neighbor U_{i-1} and its right neighbor U_{i+1}. Depending on if and where an adversary potentially disrupts the VC execution there are amount locked up differs. We analyze below all the different cases and show that no honest intermediary U_i exists, such that $\mathbf{PathBalance}(U_i)$ is lower after the execution.

1. The adversary disrupts the VC execution before it reaches U_i. In this case, U_i has no coins locked up and therefore the balance does not change.

2. The adversary disrupts the VC execution after U_i and U_{i-1} have updated their channel for opening. In this case, U_{i-1} has a non-negative amount of coins locked up with U_i. Regardless of the outcome, the balance of U_i can only increase or stay the same, since the locked-up coins come from U_{i-1}.

3. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for opening. In this case, U_{i-1} has a non-negative amount α_{i-1} of coins locked up with U_i. U_i has the same amount (minus a fee) α_i locked up with U_{i+1}.

4. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for closing. In this case, U_{i-1} has a non-negative amount α_{i-1} of coins locked up with U_i. U_i has the smaller amount α'_i locked up with U_{i+1}.

5. The adversary disrupts the VC execution after U_i and U_{i+1} have updated their channel for closing. In this case, U_{i-1} has a non-negative amount α'_{i-1} of coins locked up with U_i. U_i has the same amount (minus a fee) α'_i locked up with U_{i+1}.

To sum up, in all cases the money that U_i locks up U_{i+1} is always either the same or less than what U_{i-1} locks up with U_i. Now in each of these five cases, there are two possible things that can happen. Either $\mathsf{tx}^{\mathsf{vc}}$ is posted before $T - 3\Delta - t_{\mathsf{c}}$ or it is not. In the former case, $\mathcal{F}_{Pay}$ ensures with the Respond phase, that U_i is refunding itself, thereby keeping a neutral path balance. In the case that $\mathsf{tx}^{\mathsf{vc}}$ is not posted before $T - 3\Delta - t_{\mathsf{c}}$, U_i always gets the collateral from U_{i-1} via the Respond phase of $\mathcal{F}_{Pay}$, keeping either a neutral or positive path balance. □

Endpoint security

Intuitively, a VC protocol achieves endpoint security, if the endpoints can either enforce their VC balance on-chain or, they are compensated with an amount that is at least as large as their VC balance within an agreed-upon time. More concretely in our construction, we ensure that the sender can always enforce its VC balance on-chain. For the receiver, we ensure that either the sender puts the VC funding on-chain (allowing the receiver to enforce its balance) or, it gets the full capacity of the VC after the life time T. We extend our definition of **PathBalance**(U) for the sender U_0 and the receiver U_n. For each endpoint, this is the balance that it holds in the VC, if the VC is offloaded or 0, if the VC is not offloaded, plus its respective balance in its channel with its direct neighbor on the path.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T, such that the output of $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Then, the challenger simulates opening a VC from the OpenVC phase on input $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that a corrupted user U_i needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer, thereby giving the attacker the opportunity to stop opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

Define x_{U_0} and x_{U_n} as the latest balance of the sender and receiver in the VC, respectively. We say that the adversary wins the game if for an honest sender $\mathbf{PathBalance}(U_0)$ is lower (by an amount greater than the combined fees $(n-1)\cdot\mathsf{fee}$) after the VC execution or if for an honest receiver, $\mathbf{PathBalance}(U_n)$ is lower after T, compared balance with their respective neighbors before the VC execution plus x_{U_0} or x_{U_n}, respectively.

Definition 22 (Endpoint security)**.** We say a VC protocol achieves endpoint security if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 20 (Donner achieves endpoint security)**.** *Donner virtual channel executions achieve endpoint security as defined in Definition 22.*

Proof. For an honest sender, there are two possible scenarios. Either, $\mathcal{F}_{Pay}$ has updated (or registered an update via $\mathcal{S}$ in) the channel between U_0 and U_1 to exactly the final balance α_i' ($= x_{U_0}$ minus fees) in the CloseVC phase before the round $T - 3\Delta - t_{\mathsf{c}}$. Or, if not, $\mathcal{F}_{Pay}$ has instructed the simulator to publish $\mathsf{tx}^{\mathsf{vc}}$, allowing the balance to be enforceable on-chain. In both cases, $\mathbf{PathBalance}(U_0)$ is not lower than its initial balance with U_1 plus x_{U_0} minus the sum of all fees $(n-1)\cdot\mathsf{fee}$.

For an honest receiver, there are also two possible scenarios. Either, the VC was offloaded, allowing U_n to enforce its balance on-chain, or it is not. If VC is not offloaded, U_n either gets the full VC capacity, if the channel with U_{n-1} was not updated in the CloseVC phase or, its actual balance if it was updated in the CloseVC phase. The $\mathbf{PathBalance}(U_n)$ is therefore not lower. □

Reliability

Intuitively, we say that a VC protocol achieves reliability if after successfully opening the VC, no (colluding) malicious intermediaries can force two honest endpoints to close or offload the virtual channel before the lifespan T of the VC expires. Note that in this intuition we write before T, when technically the offloading process has to be initiated sometime before, i.e., at time $T - 3\Delta - t_{\mathsf{c}}$.

Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T, such that the output of

$\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers and then simulates opening a VC from the OpenVC phase on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that a corrupted user U_i needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer, thereby giving the attacker the opportunity to stop opening and trigger the offload and thereby refunding the collateral or let them be successful. If the opening was successful, an attacker can instruct the simulator to either perform updates, honestly close the VC, or do nothing. In the case of an honest closure, the queries to corrupted users are forwarded to the attacker, who again can let the closure be successful or force an offload.

We say that the adversary wins the game if after successfully opening the VC, i.e., the OpenVC and Finalize phases are completed successfully, the VC is offloaded before $T - 3\Delta - t_{\mathsf{c}}$.

Definition 23 (Reliability)**.** We say that a VC protocol achieves reliability if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with negligible probability.

Theorem 21 (Donner achieves reliability)**.** *Donner virtual channel executions achieve reliability as defined in Definition 23.*

Proof. This follows directly from $\mathcal{F}_{Pay}$. Note that after a successful OpenVC and Finalize phase, the only way for a VC to be offloaded is if the close phase is not reaching the sender until time $T - 3\Delta - t_{\mathsf{c}}$. □

Endpoint anonymity

A VC protocol achieves endpoint anonymity, if it achieves sender anonymity and receiver anonymity. Intuitively, we say that a VC protocol achieves *sender anonymity* if an adversary controlling an intermediary node cannot distinguish the case where the sender is its left neighbor in the path from the case where the sender is separated by one (or more) intermediaries. For receiver anonymity, an intermediary has to be unable to distinguish that the right neighbor is the receiver from the case that the intermediary and the receiver are separated by one (or more) intermediaries.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the number of intermediary nodes between the sender and the adversary. In particular, $\mathsf{channelList}_0$ is formed by U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users U_0, U_1, U^*, U_2. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a VC capacity α as well as a channel life time T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}_b$. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha +$

fee, T) to the challenger. The challenger sets $\texttt{sid}$ and $\texttt{pid}$ to two random identifiers. Moreover, the challenger picks a bit b at random and simulates the OpenVC phase on input $(\texttt{sid}, \texttt{pid}, \texttt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$, followed by the Finalize, Update and CloseVC phases. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 24 (Sender anonymity)**.** We say that a VC protocol achieves sender anonymity if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Theorem 22 (Donner achieves sender anonymity)**.** *Donner virtual channel executions achieve sender anonymity as defined in Definition 24.*

Proof. The message $(\texttt{sid}, \texttt{pid}, \texttt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$ that is sent by $\mathcal{F}_{Pay}$ to the simulator in the OpenVC phase, is leaked to the adversary. By looking at $\overline{\gamma_{i-1}}$, $\overline{\gamma_i}$ and opening onion_{i+1}, U^* knows its neighbors U_1 and U_2. We know that U^* cannot learn any additional information about the path from T, $\overline{\gamma_{i-1}}$ and $\overline{\gamma_i}$. Since the amount to be sent was increased fee for the path $\mathsf{channelList}_1$, the amount α_i for U_i is identical for both cases. This leaves $\mathsf{tx}^{\mathsf{vc}}$, rList, $\theta_{\epsilon_{i-1}}$, θ_{ϵ_i} and onion_{i+1}. Let us assume, that there exists an adversary that can break sender anonymity. There are two possible cases.

1. The adversary finds out by looking at $\mathsf{tx}^{\mathsf{vc}}$, rList, $\theta_{\epsilon_{i-1}}$ and θ_{ϵ_i}. By design, the adversary knows that outputs $\theta_{\epsilon_{i-1}}$ belongs to its left neighbor U_1 and θ_{ϵ_i} to itself. We defined that the output, that serves as input for $\mathsf{tx}^{\mathsf{vc}}$, has never been used and is unlinkable to the sender and check this in $\texttt{checkTxIn}$. Looking at the outputs of $\mathsf{tx}^{\mathsf{vc}}$, the adversary knows to whom all but one output belongs. Since our adversary breaks the sender anonymity, it needs to be able to reconstruct, to whom this final output of $\mathsf{tx}^{\mathsf{vc}}$ belongs observing rList. This contradicts our assumption of unlinkable stealth addresses.

2. The adversary finds out by looking at onion_{i+1}. The adversary controlling U^* is able to open onion_{i+1} revealing U_2, a message m and onion_{i+2}. Since our adversary breaks the sender anonymity, he has to be able to open onion_{i+2} to reveal if U_2 is the receiver or not, thereby learning who is the sender. This contradicts our assumption of secure anonymous communication networks.

These two cases lead to the conclusion, that a PPT adversary that can win the sender anonymity game with a probability non-negligibly better than $1/2$, can also break our assumptions of unlinkability of stealth addresses or secure anonymous communication networks. Note that both receiver anonymity and its proof are analogous to the sender anonymity. □

Path privacy

Intuitively, we say that a VC protocol achieves *path privacy* if an adversary controlling an intermediary node does not know what other nodes are part of the path other than its own neighbors.

A bit more formally, consider the following game. The adversary controls node U^* and selects two paths $\mathsf{channelList}_0$ and $\mathsf{channelList}_1$ that differ on the nodes other than the adversary neighbors. In particular, the path $\mathsf{channelList}_0$ is formed by U_0, U_1, U^*, U_2, U_3 whereas the path $\mathsf{channelList}_1$ contains the users $U'_0, U_1, U^*, U_2, U'_3$. Note that we force both queries to have the same path length to avoid a trivial distinguishability attack based on path length. Further note that we force that in both paths, the adversary has the same neighbors as otherwise there exists a trivial distinguishability attack based on what neighbors are used in each case.

Additionally, the adversary picks transaction $\mathsf{tx}^{\mathsf{in}}$, a VC capacity α as well as a life time T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins. Finally, the adversary sends two queries $(\mathsf{channelList}_0, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ and $(\mathsf{channelList}_1, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to two random identifiers. Moreover, the challenger picks a bit b at random and simulates the setup and open phases on input $(\mathtt{sid}, \mathtt{pid}, \mathtt{SETUP}, \mathsf{channelList}_b, \mathsf{tx}^{\mathsf{in}}, \alpha, T, \overline{\gamma_0})$. Every time that the corrupted user U^* needs to be contacted, the challenger forwards the query to the attacker and waits for the corresponding answer.

We say that the adversary wins the game if it correctly guesses the bit b chosen by the challenger.

Definition 25 (Path privacy)**.** We say that a VC protocol achieves path privacy if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Theorem 23 (Donner achieves path privacy)**.** *Donner virtual channel executions achieve path privacy as defined in Definition 25.*

Proof. As this proof is analogous to the proof for sender privacy, refer to that proof and reiterate the idea here. Again, the simulator leaks the same message $(\mathtt{sid}, \mathtt{pid}, \mathtt{open}, \mathsf{tx}^{\mathsf{vc}}, \mathsf{rList}, \mathsf{onion}_{i+1}, \alpha_i, T, \overline{\gamma_{i-1}}, \overline{\gamma_i}, \theta_{\epsilon_{i-1}}, \theta_{\epsilon_i})$ to the adversary. Again, the adversary can find out the correct bit b by looking at (i) $\mathsf{tx}^{\mathsf{vc}}$ and rList or (ii) at onion_{i+1}. If there exists an adversary that breaks the path privacy of Donner, then it also can be used to break (i) unlinkability of stealth addresses or (ii) secure anonymous communication networks. □

Value privacy

Intuitively, a VC protocol achieves value privacy, if no intermediaries gains information about the VC payments of two honest endpoints other than the opening and closing

balances of each endpoint. In particular, no intermediary learns about number of transactions being exchanged and their amount. Formally, consider the following game. The adversary selects a $\mathsf{channelList}$, a transaction $\mathsf{tx}^{\mathsf{in}}$, a virtual channel capacity α and a channel lifetime T such that the output $\mathsf{tx}^{\mathsf{in}}.\mathsf{output}[0]$ holds at least $\alpha + n \cdot \epsilon$ coins, where n is the length of the path defined in $\mathsf{channelList}$. The adversary sends the tuple $(\mathsf{channelList}, \mathsf{tx}^{\mathsf{in}}, \alpha, T)$ to the challenger.

The challenger sets `sid` and `pid` to random identifiers and simulates the opening of the virtual channel for the given parameters, forwarding queries that a corrupted intermediary would receive to the adversary. After the VC has been opened successfully, we denote the current round in the simulation as τ the challenger asks the adversary to select two lists of payments p_0 and p_1 with a length in range $[0, k]$, containing VC payments between the endpoints and their order. k denotes the maximum number of transactions that are possible within the time period between τ and when the VC needs to be honestly closed. The adversary can select arbitrary payments in an arbitrary direction with an amount between 0 and the balance of the respective sending user at the time the payment is performed. Additionally, performing either list of payments has to result in the same end balance, to avoid trivial distinction by looking at the final balance. That is, U_0's final balance is $\alpha - \alpha'$ and U_n's final balance is α', with $0 \leq \alpha' \leq \alpha$. The adversary sends p_0 and p_1 to the challenger.

The challenger picks a random bit $b \in \{0, 1\}$, and then performs the payments specified in p_b. After the payments, the challenger initiates the honest closing such, that if successful, the closing will be completed 1 round before $T - t_{\mathsf{c}} - 3\Delta$, forwarding queries to corrupted intermediaries again to the adversary. This gives the chance to the adversary, to let either VC close honestly or force to offload.

We say that an adversary wins the game, if it correctly guesses the bit b chosen by the challenger.

Definition 26 (Value privacy)**.** We say that a VC protocol achieves path value if for every PPT adversary $\mathcal{A}$, the adversary wins the aforementioned game with probability at most $1/2 + \epsilon$, where ϵ denotes a negligible value.

Theorem 24 (Donner achieves path privacy)**.** *Donner virtual channel executions achieve value privacy as defined in Definition 26.*

Proof. This property follows directly from $\mathcal{F}_{Pay}$ and $\mathcal{F}_{Channel}$. The only information regarding the VC updates is sent by either VC endpoint to $\mathcal{F}_{Pay}$ (in the Update phase) and forwarded to $\mathcal{F}_{Channel}$, other than that, the two simulations of the challenger are identical. The adversary sees only the messages that the challenger forwards to the corrupted intermediaries, which means that the adversary knows neither about the content nor the existence of these VC update messages in both scenarios. Additionally, the functionality $\mathcal{F}_{Channel}$ does not expose the internal state of a channel to anyone but the two users of it, in the case of the VC, the two endpoints.

The adversary has two options, either letting the VC close honestly or, forcing the VC to offload. In the former case, the adversary will see only the final balance α' being forwarded in the close request. In the latter case, the adversary will learn about the final balance in the VC, after it is offloaded and it is closed. It follows, that an adversary cannot guess b correctly with a probability better than $1/2 + \epsilon$, where ϵ denotes a negligible value. □