Informatics

Foundations of Adaptor Signatures for Distributed Ledger Protocols

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Erkan Tairi
Registration Number 01029450

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Matteo Maffei
Second advisor: Dr. Daniel Slamanig

The dissertation has been reviewed by:

Sebastian Faust

Foteini Baldimtsi

Vienna, February 12, 2024

Erkan Tairi

Declaration of Authorship

Dipl.-Ing. Erkan Tairi

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, February 12, 2024

Erkan Tairi

Acknowledgements

First and foremost, I would like to thank my advisors Matteo Maffei and Daniel Slamanig for giving me the opportunity to write this thesis in the first place, and for guiding me throughout my PhD. Their valuable academic guidance, enthusiasm for research and unending motivation were sources of inspiration and helped me overcome the desperate periods throughout my PhD. Moreover, I am also grateful to them for always trusting and promoting me.

I would like to also thank all (past and present) members of the Security & Privacy Research Unit at TU Wien, many of whom I consider to be good friends, for welcoming me to the group and making me feel like surrounded by family. Furthermore, I would like to thank Pedro Moreno-Sanchez for his support throughout my PhD, including applications and thesis, and for always being a great friend.

I would like to express my gratitute to all my amazing collaborators in the past years, from whom I learned immensely. In particular, I would like to thank Valerio Cini, Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Sebastian Ramacher, Clara Schneidewind, Daniel Slamanig, Christoph Striecks, Sri AravindaKrishnan Thyagarajan and Akin Ünal.

I am also grateful to Dario Fiore for hosting me for a research visit at IMDEA Software Institute and to Dennis Hofheinz for hosting me at ETH Zürich. Learning from Dario's and Dennis' creative thinking and vision for research has been enlightening.

I would like to thank my parents for their constant support and endless love, along with my sister and brother-in-law for supporting and helping me throughout my time in Vienna. I also want to thank my old friends in Macedonia for keeping me grounded whenever I visit home.

Finally, and most significantly, I would like to thank Julija Martinkutė for reminding me what are the most important things in life, which I often tend to forget, and for always loving me and being there for me (even at the times when I am annoying).

Kurzfassung

Die Herausforderungen im Hinblick auf Skalierbarkeit und Interoperabilität bei aktuellen Kryptowährungen führten zur Entwicklung kryptografischer Protokolle, welche effiziente Anwendungen auf und zwischen weit verbreiteten Kryptowährungen wie Bitcoin oder Ethereum ermöglichen. Beispiele für solche Protokolle sind (virtuelle) Zahlungskanäle, atomare Tauschgeschäfte, orakelbasierte Verträge, deterministische Wallets und Coin Mixing Services. Viele dieser Protokolle greifen lediglich auf minimale Funktionalitäten zurück, welche von zahlreichen Kryptowährungen unterstützt werden. Insbesondere haben sich Adaptor-Signaturen (AS) als effektives Werkzeug zum Entwerfen von Blockchain-Protokollen erwiesen, die weitgehend unabhängig von der spezifischen Logik der zugrunde liegenden Kryptowährung sind.

Obwohl AS-basierte Protokolle auf denselben kryptographischen Prinzipien aufbauen, sind sie im Allgemeinen weder post-quantum sicher, noch gibt es eine modulare Methode, um ihre Sicherheit zu analysieren. Stattdessen fokussieren sich sämtliche Arbeiten, die derartige Protokolle untersuchen, auf das wiederholte Beweisen, wie Adapter-Signaturen zur kryptografischen Verknüpfung von Transaktionen verwendet werden. Hierbei werden stark vereinfachte Blockchain-Modelle berücksichtigt, die keine sicherheitsrelevanten Aspekte der Transaktionsausführung im Konsens der Blockchain abbilden.

In dieser Arbeit entwickeln wir ein Post-Quantum-AS-Schema, das auf kryptographischen Standardannahmen über Isogenien basiert. Wir belegen formal die Sicherheit unserer Konstruktion im (Quanten-) Random Oracle Model. Anschließend behandeln wir AS im Rahmen des Universal Composability (UC) Frameworks, um die Modularität von AS zu ermöglichen.

Darüber hinaus präsentieren wir LedgerLocks, ein Framework für das sichere Design von AS-basierten Blockchain-Anwendungen in Gegenwart einer realistischen Blockchain. LedgerLocks führt das Konzept der AS-gesperrten Transaktionen ein, welche Transaktionen sind, deren Veröffentlichung an die Kenntnis eines kryptographischen Geheimnisses gebunden ist. Wir argumentieren, dass AS-gesperrte Transaktionen gemeinsame Bausteine von AS-basierten Blockchain-Protokollen sind. Gleichzeitig definieren wir mit $\mathcal{G}_{\mathsf{LedgerLocks}}$ ein realistisches Ledger-Modell im UC-Framework, das über eingebaute Unterstützung für AS-gesperrte Transaktionen verfügt. Durch die Abstraktion von LedgerLocks von der kryptographischen Realisierung von AS-gesperrten Transaktionen können sich Protokolldesigner stattdessen auf die spezifischen Sicherheitsüberlegungen in der Blockchain konzentrieren.

Wir zeigen schließlich die Verwendung von LedgerLocks bei der Modellierung und dem Nachweis der Sicherheit von AS-basierten Blockchain-Protokollen, indem wir eine Zahlungskanal-Konstruktion und eine darauf aufbauende datenschutzfreundliche Zahlungskanal-Hub-Konstruktion vorstellen.

Abstract

The scalability and interoperability challenges in current cryptocurrencies have motivated the design of cryptographic protocols that enable efficient applications on top and across widely used cryptocurrencies such as Bitcoin or Ethereum. Examples of such protocols include (virtual) payment channels, atomic swaps, oracle-based contracts, deterministic wallets, and coin mixing services. Many of these protocols are built upon minimal core functionalities supported by a wide range of cryptocurrencies. Most prominently, adaptor signatures (AS) have emerged as a powerful tool for constructing blockchain protocols that are (mostly) agnostic to the specific logic of the underlying cryptocurrency.

Even though AS-based protocols are built upon the same cryptographic principles, they in general are neither post-quantum secure nor there exists a modular way to reason about their security. Instead, all the works analyzing such protocols focus on reproving how adaptor signatures are used to cryptographically link transactions while considering highly simplified blockchain models that do not capture security-relevant aspects of transaction execution in blockchain-based consensus.

In this thesis, we construct a post-quantum AS scheme that relies on standard cryptographic assumptions on isogenies, and we formally prove the security of our construction in (quantum) random oracle model. Then, we provide a composable treatment of AS within the Universal Composability (UC) framework to facilitate modularity of AS.

Moreover, we present LedgerLocks, a framework for the secure design of AS-based blockchain applications in the presence of a realistic blockchain. LedgerLocks defines the concept of AS-locked transactions, transactions whose publication is bound to the knowledge of a cryptographic secret. We argue that AS-locked transactions are the common building block of AS-based blockchain protocols and we define $\mathcal{G}_{\mathsf{LedgerLocks}}$, a realistic ledger model in the UC framework with built-in support for AS-locked transactions. As LedgerLocks abstracts from the cryptographic realization of AS-locked transactions, it allows protocol designers to focus on the blockchain-specific security considerations instead.

Finally, we showcase the usage of LedgerLocks in modeling and proving security of AS-based blockchain protocols by presenting a payment channel construction and a privacy-preserving payment channel hub (PCH) construction built on top of it.

Main Publications and Contributions

This thesis is based on the following publications:

[TMS23] Erkan Tairi, Pedro Moreno-Sanchez, and Clara Schneidewind. LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures. In *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 859-873, Copenhagen, Denmark, November 26-30, 2023. ACM Press.

[GMMMTT22] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri AravindKrishnan Thyagarajan. Foundations of Coin Mixing Services. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1259-1273, Los Angeles, CA, USA, November 7-11, 2022. ACM Press.

[TMM21a] 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*, pages 1834–1851, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press

[TMM21b] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. Post-Quantum Adaptor Signature for Privacy-Preserving Off-Chain Payments. In Nikita Borisov and Claudia Díaz, editors, *FC 2021: 25th International Conference on Financial Cryptography and Data Security, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 131–150, Virtual Event, March 1–5, 2021. Springer, Heidelberg, Germany.

To fit the above publications into the present thesis, I provide a unified introduction in Chapter 1, and unified the notations and models of the cryptographic primitives in Chapter 2. The results of the above publications are then rearranged and presented in Chapters 3 to 6 as described below. Finally, I provide a conclusion and future research directions in Chapter 7.

Research projects are often highly collaborative, which makes it very difficult or even impossible to split the final result into individual contributions and assign authorship to each of them. The works underlying this thesis are no exception. Presented research results developed through numerous fruitful discussions with my co-authors. In the following, I aim to specify my contribution to the works underlying this thesis, where possible.

Chapter 3 is based on the joint work [TMM21b] with Matteo Maffei and Pedro Moreno-Sanchez. I designed the post-quantum adaptor signature scheme from isogenies and proved its security in the (quantum) random oracle model. Moreover, I also provided an implementation of the construction and performed the benchmarking. On the other hand, Pedro Moreno-Sanchez and Matteo Maffei worked on providing concrete applications of our post-quantum adaptor signature, such as a post-quantum multi-hop payment protocol, which is not part of this thesis.

Chapters 4 and 5 are based on the joint work [TMSS23] with Pedro Moreno-Sanchez and Clara Schneidewind. I developed the ideal functionalities and protocols for cryptographic conditions and adaptor signatures, as presented in Chapter 4, and provided their security proof with the Universal Composability (UC) framework.

Similarly, the LedgerLocks framework, given in Chapter 5, which includes the lock-enabling ledger functionality and its corresponding protocol, was mostly designed and proven secure with the UC framework by myself. Clara Schneidewind was responsible for the analysis of state-of-the-art in blockchain protocols based on adaptor signatures, along with providing a template for using our framework and showcasing an atomic swap protocol within our framework. Pedro Moreno-Sanchez provided additional blockchain protocols within our framework, such as a payment channel and multi-hop payment constructions.

Chapter 6 is (mainly) based on previously published joint works [TMSS23, TMM21a, GMM^{+}22]. More precisely, Section 6.1 extends the partial payment channel construction given in the joint work [TMSS23] with Pedro Moreno-Sanchez and Clara Schneidewind. This partial payment channel construction was initially written down by Pedro Moreno-Sanchez, however, it lacked an ideal functionality and its corresponding UC realization proof. In this thesis, I filled these gaps by providing a concrete ideal functionality and security proof for the payment channel construction within our LedgerLocks framework. Clara Schneidewind worked out the concrete timelocks needed for secure realization of our payment channel construction.

On the other hand, Section 6.2 is based on two joint works [TMM21a] and [GMM^{+}22] with Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, and Sri AravindKrishnan Thyagarajan. More precisely, [GMM^{+}22] presents a potential security flaw in [TMM21a] and supersedes that work. Additionally, [GMM^{+}22] provides blind conditional signatures, a modular building block for payment channel hub (PCH) constructions, and provides constructions of PCH that are provably secure in both game-based setting and within the UC framework. Giulio Malavolta, Matteo Maffei, and Pedro

Moreno-Sanchez identified the flaw in [TMM21a]. Sri AravindKrishnan Thyagarajan and Noemi Glaeser mostly developed the notion of blind conditional signatures and provided new PCH constructions along with security proofs. I helped in the development of the constructions and performed the performance evaluation. This thesis only present the UC secure PCH construction from [GMM$^+$22], albeit it is re-written and re-proven secure using our LedgerLocks framework.

Other Selected Publications

Some of the other publications not covered by this thesis, written in part by the author of this thesis during his doctoral study, are listed below:

[TÜ24] Erkan Tairi, and Akın Ünal. Lower Bounds for Lattice-based Compact Functional Encryption. In *EUROCRYPT 2024* (to appear).

[CRSST24] Valerio Cini, Sebastian Ramacher, Daniel Slamanig, Christoph Striecks, and Erkan Tairi. (Inner-Product) Functional Encryption with Updatable Ciphertexts. In *Journal of Cryptology* 2024.

[GRSSTZ23] Christian Göth, Sebastian Ramacher, Daniel Slamanig, Christoph Striecks, Erkan Tairi, and Alexander Zikulnig. Optimizing 0-RTT Key Exchange with Full Forward Security. In *CCSW '23: Cloud Computing Security Workshop 2023*, pages 55-68, Copenhagen, Denmark, November 26, 2023. ACM Press.

[CRSST21] Valerio Cini, Sebastian Ramacher, Daniel Slamanig, Christoph Striecks, and Erkan Tairi. Updatable Signatures and Message Authentication Codes. In Juan Garay, editors, *PKC 2021: 24th International Conferences on Theory and Practice of Public Key Cryptography, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 691-723, Virtual Event, May 10-13, 2021. Springer, Heidelberg, Germany.

Contents

CHAPTER 1

Introduction

Blockchain-based cryptocurrencies such as Bitcoin, enable mutually distrusting users to perform financial transactions without relying on a trusted third party. However, for their large-scale adoption, cryptocurrencies face major interoperability and scalability challenges. These challenges can be tackled with the help of cryptographic protocols that form a more flexible application layer on top of the core cryptocurrency functionalities. Prominent examples are atomic swaps [TMM22] for users to trade their coins across different cryptocurrencies or payment channels [AEE+21] to perform an unlimited number of fast bilateral payments while publishing only a small number of transactions on the blockchain.

To ease the interoperability across cryptocurrencies, these protocols are usually realized upon simple core operations supported by most cryptocurrencies (e.g., payment authorization with a digital signature from the sender). This endeavor has been facilitated by the recent discovery of *adaptor signatures (AS)* [AEE+21, EFH+21], which allow for conditioning the creation of a digital signature on the knowledge of a cryptographic secret.

More precisely, AS can be seen as an extended form of a standard digital signature, where one can create a “pre-signature” that can be converted into a (full) signature with respect to an instance of a hard relation R (e.g., the discrete logarithm relation R_{DL}, i.e., $(Y, y) \in R_{\mathsf{DL}} \iff Y = g^y$). The resulting signature can then be verified by the miners using the standard verification algorithm from the digital signature scheme. AS provide the following two intuitive properties: (i) only the user knowing the witness of the hard relation can convert the pre-signature into a valid signature; and (ii) anybody with access to the pre-signature and the corresponding signature can extract the witness of the hard relation.

This building block has been shown highly useful in practice to build off-chain payment applications such as (generalized) payment channels [AEE+21], payment-channel networks [MMS$^+$19], payment channel hubs [TMM21a, GMM$^+$22, QPM$^+$23], decentralized oracle contracts [MTV$^+$22] and many others, being adopted in real-world blockchain protocols, such as the Lightning Network, the COMIT Network, ZengoX and others.

Despite the ubiquity of AS, post-quantum security and simulation security of AS have not been properly studied before. Moreover, there exists no modular way to reason about the security of the existing AS-based blockchain protocols. We highlight these drawbacks in more detail below and give our contributions in Section 1.1.

Post-Quantum AS. We can construct AS generically from digital signature schemes obtained by identification schemes via the Fiat-Shamir transform [FS87], as shown by Erwig et al. [EFH$^+$21]. However, their transformation are not (fully) suitable for post-quantum constructions. For example, lattice-based signature schemes are obtained using rejection sampling, i.e., via the Fiat-Shamir-with-Aborts methods [Lyu09], however, the generic transformation of [EFH$^+$21] does not account for these potential aborts. Moreover, isogeny-based signature schemes are usually based on cryptographic group actions [ADMP20], which provides limited algebraic operations, and hence, does not directly fit the framework of Erwig et al. [EFH$^+$21].

Therefore, given the shortcomings of transformation given by Erwig et al. [EFH$^+$21] and the relevance in practice of AS, there is a need to design post-quantum instances of AS. For instance, there exist several efforts from NIST to standardize quantum resistant digital signatures[1]. The blockchain community has also shown interest in migrating towards post-quantum secure alternatives. For example, Ethereum is planing[2] to have an option for a post-quantum signature and Zcash developers plan to update their protocol with post-quantum alternatives[3].

The first seminal contribution in this direction was by Esgin et al. [EEE20], who proposed the first instance of a post-quantum AS, called LAS, which is based on the standard lattice assumptions, such as Module-SIS and Module-LWE. This construction, however, presents a few limitations with regard to the correctness, communication overhead, and privacy. From the correctness point of view, LAS requires usage of two hard relations, R and R', where R is the base relation and R' is the extended relation that defines the relation for extracted witnesses. The reason for this is due to the inherent knowledge/soundness gap in lattice-based zero-knowledge proofs [LNP22]. Hence, as mentioned by the authors, LAS only achieves *weak* pre-signature adaptability, which guarantees that only the statement/witness pairs satisfying R are adaptable, and not those satisfying R'. In practice, this implies that the applications that use LAS as a building block require a zero-knowledge proof to guarantee that the extracted witness is of sufficiently small norm and belongs to the base relation R, which in turn guarantees that the pre-signature adaptability would work. However, the currently most efficient variant of such a proof has size of more than 10KB [LNP22], which would incur significant (off-chain) communication overhead to the applications using LAS.

[1] https://csrc.nist.gov/projects/pqc-dig-sig/round-1-additional-signatures

[2] https://ethereum.org/en/roadmap/future-proofing

[3] https://github.com/zcash/zcash/issues/6121

From the privacy point of view, when LAS is used inside certain applications, such as building payment-channel networks (PCNs), it can leak non-trivial information that hinders the privacy of the overall construction. In a nutshell, the reason for that is that the witness for adapting the pre-signature in LAS is a vector whose infinity norm is 1. Privacy-preserving applications, such as PCNs, require to encode a randomization factor at each hop, which in LAS is encoded by adding a new vector whose infinity norm is 1 for each hop [EEE20, Section 4.2]. However, this leads to a situation where a node at position k in the payment path receives a vector with infinity norm k with high probability, learning at least how many parties are before it on the path. Moreover, if an intermediary observes that the norm is 1, then it knows that (with high probability) the party before it is the sender. Encoding a vector of random but small norm (i.e., padding) for each hop does not help either, as each sender-receiver pair would use a unique norm, thus breaking relationship anonymity.

Finally, given the ongoing standardizations efforts of NIST, it is useful to have several candidates of quantum-resistant AS building upon different cryptographic assumptions to aid the related discussion (e.g., if one assumption gets broken, we may still have standing post-quantum constructions).

Security Analysis of AS-Based Blockchain Protocols. Despite the multitude of cryptographic blockchain protocols relying on adaptor signatures [MMS+19, AEE+21, AME+21, TMM21a, Mir22, ATM+22, LGKK22, MTV+22, ER22, TMM22, GMM+22, BM22, QPM+23], the security analysis of these protocols are either not modular or incomplete.

More precisely, all these AS-based blockchain protocols are proven secure in the Universal Composability framework of Canetti [Can20]. However, there exists no adaptor signature functionality that one can use to modularly describe and prove these protocols. This in turn means that during the security proof of these protocols one needs to reprove how adaptor signatures are used to cryptographically link transactions by doing reductions to the properties of AS, hence, sacrificing modularity.

Moreover, these protocols not only rely on the correct usage of cryptographic primitives used in the message exchanges between protocol participants but also on the guarantees that stem from the underlying blockchain consensus. In spite of that, all current works proposing new AS-based blockchain protocols study their security in the context of highly simplified ledger models, defined in an ad-hoc manner [MMS+19, AMKM21a, AMKM21b, AEE+21, TMM21a, TMM22, GMM+22, QPM+23].

However, the subtleties of the ledger model have a significant influence on the blockchain protocol security and neglecting these aspects can easily result in undetected security issues as we show in Section 5.1. Consequently, it is highly desirable to build an infrastructure that facilitates the reasoning about AS-based blockchain protocols in the presence of a realistic ledger. Such an infrastructure should ideally separate the reasoning about ledger-specific aspects of AS-based blockchain protocols from the cryptographic operations. We observe that adaptor signatures are used in these protocols to encode a generic building block that we call here *AS-locked transactions*. AS-locked transactions are transactions whose publication on the blockchain is bound to the knowledge of a cryptographic secret in two ways:

1. knowing the cryptographic secret is a prerequisite for a party holding the AS-locked transaction to publish it on the ledger; and

2. the publication of the AS-locked transaction on the ledger reveals the secret to all parties holding the AS-locked transaction.

By synthesizing this building block and integrating it into a realistic ledger functionality, we can describe many blockchain protocols in terms of this functionality without the cryptographic interactions between the protocol participants.

In the state of the art, AS-based blockchain protocols are mainly given through the exchange of cryptographic messages among the participants. These interactions shall ensure that the protocol participants can construct valid transactions to be published on the blockchain (given through a simplified ledger functionality $\mathcal{G}_L$) in compliance with the protocol goals. The cryptographic reasoning for showing the security of these interactions is essentially the same throughout the state-of-the-art protocols.

Therefore, by defining a realistic ledger functionality that supports generic AS-locked transactions, and thus, subsumes the cryptographic aspects of these protocols, we can describe AS-based blockchain protocols in terms of AS-locked transactions without further need for cryptographic interactions between the protocol participants. Consequently, the subsequent security analysis of such protocols would not require cryptographic reasoning but could focus on the ledger-specific security arguments. Lifting the burden of concurrently reasoning about both cryptographic and ledger-specific security aspects paves the ground for the security analysis of blockchain protocols in realistic ledger models.

However, constructing such a realistic ledger functionality that supports AS-locked transactions comes with multiple technical challenges. First, the logic of protocols using AS-locked transactions usually relies on relating the transactions through the structure of their cryptographic conditions. Therefore, for a truly modular reasoning we need a general model of cryptographic conditions that integrates with the ledger functionality and is adaptable to the protocol needs. Second, to show that such a ledger functionality is realizable by adaptor signatures in the presence of cryptographic conditions, a novel composable notion of adaptor signature security is needed. Finally, to facilitate flexible reasoning in a faithful ledger model, we need to model that the ledger functionality exposes provably realistic ledger behavior while supporting a generic notion of AS-locked transactions.

1.1 Contributions

This thesis aims at providing theoretical foundations of AS and blockchain protocols built on top of them in order to tackle the previously described challenges. To this end, we describe a post-quantum AS construction, formalize AS in the Universal Composability (UC) framework and describe a novel framework to modularize and ease the reasoning about security of AS-based blockchain protocols.

1.1.1 Post-Quantum Adaptor Signatures

In Chapter 3 we give a construction of post-quantum AS that preserves the security and privacy guarantees required by off-chain application.

To this end, we design IAS, a construction for AS that builds upon the post-quantum signature scheme CSI-FiSh [BKV19], and relies on hardness of standard cryptographic assumptions from isogenies. We formally prove the security of IAS in (quantum) random oracle model.

Additionally, we provide an optimized variant of IAS, for which we provide a parallelized implementation and evaluate its performance, showing that it requires ∼1500 bytes of storage on-chain (with a parameter set optimized for lower combined public key and signature size) and 140 milliseconds to verify a signature on average (i.e., the computation time for miners). We compare our construction with LAS [EEE20], a lattice-based AS scheme, and observe that our on-chain storage size is $3x$ smaller than LAS while requiring higher computation time.

1.1.2 Universally Composable Adaptor Signatures

In Chapter 4 we modularize AS by formalizing them within the UC framework of Canetti [Can20].

For this purpose, we first model cryptographic conditions as a standalone (global) ideal functionality $\mathcal{G}_{\mathsf{Cond}}$, which encodes operations over conditions, such as their composition. We define a protocol $\Pi^{R_{\mathsf{DL}}}_{\mathsf{Cond}}$, which realizes $\mathcal{G}_{\mathsf{Cond}}$ for simple (individual) conditions, merged conditions and 1-out-of-n conditions. We note that $\mathcal{G}_{\mathsf{Cond}}$ can be easily extended to account for other operations in a modular fashion, that is, without modifying the several other functionalities using it in a shared manner to keep conditions consistent across them.

Next, we model (two-party) adaptor signatures as an ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ and prove that it is UC-realized by any two-party adaptor signature with aggregatable public keys generated from an identification scheme as defined by Erwig et al. [EFH$^+$21], a class encompassing all the digital signatures used in current AS-based applications. This realization makes use of the cryptographic conditions functionality $\mathcal{G}_{\mathsf{Cond}}$ in order to check the correctness of the conditions used within $\mathcal{F}_{\mathsf{AdaptSig}}$.

1.1.3 Security Framework for Protocols Based on Adaptor Signatures

In Chapter 5 we present a novel framework, named LedgerLocks, by defining $\mathcal{G}_{\mathsf{LedgerLocks}}$, an ideal functionality that models a realistic ledger with generic AS-locked transactions. $\mathcal{G}_{\mathsf{LedgerLocks}}$ is based on the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17], which has been proven to be realizable by the Bitcoin backbone protocol [BMTZ17] as well as the proof of stake-based protocol Ouroboros Genesis [BGK+18]. We also provide a protocol $\Pi_{\mathsf{LedgerLocks}}$ that UC-realizes $\mathcal{G}_{\mathsf{LedgerLocks}}$ in the presence of $\mathcal{G}_{\mathsf{Ledger}}$ from [BMTZ17] and our AS ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$.

Finally, in Chapter 6 we demonstrate the flexibility of our framework, by using it to describe a payment channel protocol Π_{Channel} and a payment channel hub (PCH) protocol $\Pi_{\mathsf{A^2L}}$, both of them protocols relying on AS-locked transactions.

To this end, we instantiate $\mathcal{G}_{\mathsf{LedgerLocks}}$ with support for UTXO-based transaction and timelocks, which are crucial for payment channel and PCH security. The description of Π_{Channel} follows along the lines of the generalized channels construction given by Aumayr et al. [AEE+21], and it does not involve any additional cryptography, apart from AS. Therefore, we do not need to consider any cryptographic reductions and instead can focus on the delicate task of adjusting the protocol timelocks to provide security in the presence of realistic blockchains as modeled by our functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$. On the other hand, our PCH construction $\Pi_{\mathsf{A^2L}}$ makes use of a commitment scheme, a linearly homomorphic encryption scheme and a non-interactive zero-knowledge proof system, in addition to AS, in order to provide certain privacy guarantees (we detail this in Section 6.2). Since AS part is already encapsulated via our functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$, we only need to focus on the security properties of the other cryptographic primitives, which again simplifies the security proof.

CHAPTER 2

Preliminaries

Notation. We denote the security parameter by $\lambda \in \mathbb{N}$, by which each scheme and adversary is parameterized. For $n \in \mathbb{N}$, set $[n] = \{1, 2, \ldots, n\}$. We denote by $x \leftarrow\!\!\$\ \mathcal{X}$ the uniform sampling of the variable x from the set $\mathcal{X}$. We write $x \leftarrow \mathsf{A}(y)$ to denote that a probabilistic polynomial time (PPT) algorithm A on input y, outputs x. We use the same notation also for the assignment of the computational results, for example, $s \leftarrow s_1 + s_2$. If A is a deterministic polynomial time (DPT) algorithm, we use the notation $x := \mathsf{A}(y)$. We use the same notation for the projection of tuples, e.g., we write $\sigma := (\sigma_1, \sigma_2)$ for a tuple σ composed of two elements σ_1 and σ_2. We define *polynomial* and *negligible* functions as follows,

$$\begin{aligned} \mathsf{poly}(\lambda) &:= \left\{ p \colon \mathbb{N} \to \mathbb{N} \mid \exists d \in \mathbb{N} \colon p(\lambda) \in O(\lambda^d) \right\}, \\ \mathsf{negl}(\lambda) &:= \left\{ \varepsilon \colon \mathbb{N} \to [0,1] \mid \forall d \in \mathbb{N} \colon \limsup_{\lambda \to \infty} \varepsilon(\lambda) \cdot \lambda^d = 0 \right\}. \end{aligned}$$

Throughout this thesis we implicitly assume that negligible functions are negligible in the security parameter (i.e., $\mathsf{negl}(\lambda)$).

2.1 Mathematical Preliminaries

2.1.1 Elliptic Curves and Isogenies

Let E be an elliptic curve over a finite field $\mathcal{F}_p$ with p a large prime, and let $\mathbf{0}_E$ be the point at infinity on E. An elliptic curve is called supersingular iff its number of rational points satisfies $\#E(\mathcal{F}_p) = p + 1$. Otherwise, an elliptic curve is called ordinary. We note that in this work we are considering supersingular curves.

An isogeny between two elliptic curves E and E' is a rational map $\phi \colon E \to E'$, such that $\phi(\mathbf{0}_E) = \mathbf{0}_{E'}$, and which is also a homomorphism with respect to the natural group structure of E and E'. An isomorphism between two ellliptic curves is an injective isogeny. The j-invariant of an elliptic curve, which is a simple algebraic expression in the coefficients of the curve, is an algebraic invariant under isomorphism (i.e., isomorphic curves have the same j-invariant). As isogenies are group homomorphisms, any isogeny comes with a subgroup of E, which is its kernel. Any subgroup $S \subset E(\mathcal{F}_{p^k})$ yields a unique (up to automorphism) separable isogeny $\phi\colon E \to E/S$ with $\ker \phi = S$.

The ring of endomorphisms $\mathrm{End}(E)$ consists of all isogenies from E to itself, and $\mathrm{End}_{\mathcal{F}_p}(E)$ denotes the ring of endomorphisms defined over $\mathcal{F}_p$. For an ordinary curve $E/\mathcal{F}_p$ we have that $\mathrm{End}(E) = \mathrm{End}_{\mathcal{F}_p}(E)$, but for a supersingular curve over $\mathcal{F}_p$ we have a strict inclusion $\mathrm{End}_{\mathcal{F}_p}(E) \subsetneq \mathrm{End}(E)$. In particular, for supersingular elliptic curves the ring $\mathrm{End}(E)$ is an order of a quarternion algebra defined over $\mathbb{Q}$, while $\mathrm{End}_{\mathcal{F}_p}(E)$ is isomorphic to an order of the imaginary quadratic field $\mathbb{Q}(\sqrt{-p})$. We will identify $\mathrm{End}_{\mathcal{F}_p}(E)$ with the isomorphic order which we will denote by $\mathcal{O}$.

The ideal class group of $\mathcal{O}$ is the quotient of the group of fractional invertible ideals in $\mathcal{O}$ by the principal fractional invertible ideals, and will be denoted as $\mathrm{Cl}(\mathcal{O})$. There is a natural action of the class group on the class of elliptic curves defined over $\mathcal{F}_p$ with order $\mathcal{O}$. Given an ideal $\mathfrak{a} \subset \mathcal{O}$, we can consider the subgroup defined by the intersection of the kernels of the endomorphisms in $\mathfrak{a}$, more precisely, $S_{\mathfrak{a}} = \cap_{\alpha\in\mathfrak{a}} \ker \alpha$. As this is a subgroup of E, we can divide out by $S_{\mathfrak{a}}$ and get the isogenous curve $E/S_{\mathfrak{a}}$, which we denote by $\mathfrak{a} \star E$. This isogeny is well-defined and unique up to $\mathcal{F}_p$-isomorphism and the group $\mathrm{Cl}(\mathcal{O})$ acts via the operator $\star$ on the set $\mathcal{E}$ of $\mathcal{F}_p$-isomorphism classes of elliptic curves with $\mathcal{F}_p$-rational endomorphism ring $\mathcal{O}$. One can show that $\mathrm{Cl}(\mathcal{O})$ acts freely and transitively on $\mathcal{E}$ (i.e., $\mathcal{E}$ is a principal homogeneous space for $\mathrm{Cl}(\mathcal{O})$).

Notation. Following [BKV19], we see $\mathrm{Cl}(\mathcal{O})$ as a cyclic group with generator $\mathfrak{g}$, and we write $\mathfrak{a} = \mathfrak{g}^a$ with a random in $\mathbb{Z}_N$ for $N = \#\,\mathrm{Cl}(\mathcal{O})$ the order of the class group. We write $[a]$ for $\mathfrak{g}^a$ and $[a]E$ for $\mathfrak{g}^a \star E$. We note that under this notation $[a][b]E = [a+b]E$.

2.1.2 Computational Assumptions

The main hardness assumption underlying group actions based on isogenies is that it is hard to invert the group action.

Definition 1 (Group Action Inverse Problem (GAIP) [DG19]). *Given two elliptic curves E and E' over the same finite field and with $\mathrm{End}(E) = \mathrm{End}(E') = \mathcal{O}$, find an ideal $\mathfrak{a} \subset \mathcal{O}$ such that $E' = \mathfrak{a} \star E$.*

The CSI-FiSh signature scheme (see Section 3.1) relies on the hardness of random instance of a multi-target version of GAIP, called MT-GAIP. In [DG19] it is shown that MT-GAIP reduces tightly to GAIP when the class group structure is known (which is the case for CSI-FiSh).

Definition 2 (Multi-Target GAIP (MT-GAIP) [DG19]). *Given k elliptic curves $E_1, \ldots, E_k$ over the same field, with $\mathrm{End}(E_1) = \cdots = \mathrm{End}(E_k) = \mathcal{O}$, find an ideal $\mathfrak{a} \subset \mathcal{O}$ s.t. $E_i = \mathfrak{a} \star E_j$ for some $i, j \in \{0 \ldots, k\}$ with $i \neq j$.*

The best known classical algorithm to solve the GAIP (and in this case also the MT-GAIP) has time complexity $O(\sqrt{N})$, where $N = \#\,\mathrm{Cl}(\mathcal{O})$. On the other hand, the best known quantum algorithm is Kuperberg's algorithm for the hidden shift problem [Kup05, Kup13]. It has a subexponential complexity with the concrete security estimates still being an active area of research [BS20, Pei20]. We also remark that CSI-FiSh is not affected by the recent devastating attacks on SIDH [CD23, Rob23].

2.2 Cryptographic Preliminaries

We review the cryptographic primitives of interest in this section.

2.2.1 Non-Interactive Zero-Knowledge Proofs

We first recall the definition of a hard relation.

Definition 3 (Hard Relation)**.** *Let R be a relation with statement/witness pairs (Y, y). Let us denote L the associated language defined as $L := \{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a* hard relation *if the following holds:*

- *There exists a* PPT *sampling algorithm* $\mathsf{GenR}(1^\lambda)$ *that on input the security parameter λ outputs a statement/witness pair $(Y, y) \in R$.*
- *The relation is poly-time decidable.*
- *For all* PPT *adversaries $\mathcal{A}$ there exists a negligible function* negl*, such that:*

$$\Pr\left[(Y, y^*) \in R \;\middle|\; \begin{array}{c}(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)\\ y^* \leftarrow \mathcal{A}(Y)\end{array}\right] \leq \mathsf{negl}(\lambda),$$

where the probability is taken over the randomness of GenR *and $\mathcal{A}$.*

Let R be a binary relation as defined above, with L the language consisting of statements in R. A non-interactive zero-knowledge (NIZK) proof system [BFM88] for a language L allows proving in a non-interactive manner that some statements are in L without leaking information about the corresponding witnesses. We formally define it as follows.

Definition 4 (Non-Interactive Zero-Knowledge Proof System with Online Extractor)**.** *A non-interactive zero-knowledge proof of knowledge (NIZKPoK) Π_{NIZK} for a language $L \in \mathsf{NP}$ (with witness relation R) with an online extractor (in the random oracle model) is a tuple of algorithms $\Pi_{\mathsf{NIZK}} = (\mathsf{PGen}, \mathsf{P}, \mathsf{V})$, such that:*

$\mathsf{PGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter 1^λ, outputs a common reference string* crs.

$\mathsf{P}(\mathsf{crs}, x, w)$**:** *is a* PPT *algorithm that on input a common reference string* crs*, a statement x and a witness w, outputs a proof π.*

$\mathsf{V}(\mathsf{crs}, x, \pi)$**:** *is a* DPT *algorithm that on input a common reference string* crs*, a statement* x *and a proof* π*, outputs a bit* b*.*

We require Π_{NIZK} *to meet the following properties:*

Completeness. *For every* $(x, w) \in R$ *we have that*

$$\Pr\Big[\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda), \pi \leftarrow \mathsf{P}(\mathsf{crs}, x, w) \colon \mathsf{V}(\mathsf{crs}, x, \pi) = 1\Big] = 1.$$

Soundness. *For every* $x \notin L$*, and every adversary* $\mathcal{A}$*, we have that*

$$\Pr\Big[\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda), \pi \leftarrow \mathcal{A}(\mathsf{crs}, x) \colon \mathsf{V}(\mathsf{crs}, x, \pi) = 1\Big] \leq \mathsf{negl}(\lambda).$$

Zero-Knowledge. *There exists a* PPT *algorithm* $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ *such that for every* PPT *adversary* $\mathcal{A}$*,*

$$\begin{aligned} \Big| \Pr\Big[&\mathsf{crs} \leftarrow \mathsf{PGen}(1^\lambda) \colon \mathcal{A}^{\mathsf{P}(\mathsf{crs}, \cdot, \cdot)}(\mathsf{crs}) = 1\Big] \\ - \Pr\Big[&(\mathsf{crs}, \tau) \leftarrow \mathcal{S}_1(1^\lambda) \colon \mathcal{A}^{\mathcal{O}(\mathsf{crs}, \tau, \cdot, \cdot)}(\mathsf{crs}) = 1\Big]\Big| \leq \mathsf{negl}(\lambda), \end{aligned}$$

where $\mathcal{O}(\mathsf{crs}, \tau, \cdot, \cdot)$ *is an oracle that outputs* $\perp$ *on input* (x, w) *when* $(x, w) \notin R$ *and outputs* $\pi \leftarrow \mathcal{S}_2(\mathsf{crs}, \tau, x)$ *when* $(x, w) \in R$*.*

Online Extractor. *There exists a* PPT *algorithm* $\mathcal{K}$*, the online extractor, such that the following holds for any adversary* $\mathcal{A}$*. Let* $\mathcal{H}$ *be a random oracle,* $(x, \pi) \leftarrow \mathcal{A}^{\mathcal{H}}(\lambda)$ *and* $\mathcal{Q}_{\mathcal{H}}(\mathcal{A})$ *be the sequence of queries of* $\mathcal{A}$ *to* $\mathcal{H}$ *and* $\mathcal{H}$*'s answers. Let* $w \leftarrow \mathcal{K}(x, \pi, \mathcal{Q}_{\mathcal{H}}(\mathcal{A}))$*. Then, the following holds,*

$$\Pr\Big[(x, w) \notin R \wedge \mathsf{V}^{\mathcal{H}}(x, \pi) = 1\Big] \leq \mathsf{negl}(\lambda).$$

2.2.2 Canonical Identification Scheme

We recall the notion of canonical identification scheme [KMP16], which can be transformed to a digital signature using Fiat-Shamir heuristic [FS87].

Definition 5 (Canonical Identification Scheme [KMP16])**.** *A canonical identification scheme consists of four algorithms* $\mathsf{ID} = (\mathsf{IGen}, \mathsf{P}, \mathsf{ChSet}, \mathsf{V})$*, where*

$\mathsf{IGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* 1^λ *outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*. We assume that* pk *defines the challenge set* ChSet*.*

P**:** *is a* PPT *algorithm composed of* P_1 *and* P_2*:*

- $\mathsf{P}_1(\mathsf{sk})$*: on input a secret key* sk*, outputs a commitment* $R \in \mathcal{D}_{\mathsf{rand}}$ *and a state* st.
- $\mathsf{P}_2(\mathsf{sk}, R, h, \mathsf{st})$*: on input a secret key* sk*, commitment* $R \in \mathcal{D}_{\mathsf{rand}}$*, challenge* $h \in \mathsf{ChSet}$ *and state* st*, outputs a response* $s \in \mathcal{D}_{\mathsf{resp}}$.

$\mathsf{V}(\mathsf{pk}, R, h, s)$**:** *is a* DPT *algorithm that on input a public key* pk*, and conversation transcript composed of* (R, h, s)*, outputs a bit* b*.*

We require that for all $(\mathsf{sk}, \mathsf{pk}) \in \mathsf{IGen}(1^\lambda)$*, all* $(R, \mathsf{st}) \in \mathsf{P}_1(\mathsf{sk})$*, all* $h \in \mathsf{ChSet}$ *and all* $s \in \mathsf{P}_2(\mathsf{sk}, R, h, \mathsf{st})$*, we have that* $\mathsf{V}(\mathsf{pk}, R, h, s) = 1$*.*

2.2.3 Digital Signature

We first recall the definition and security notions of a digital signature.

Definition 6 (Digital Signature Scheme)**.** *A digital signature scheme consists of three algorithms* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sig}, \mathsf{Ver})$ *defined as follows:*

$\mathsf{KeyGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* 1^λ*, outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$*.*

$\mathsf{Sig}(\mathsf{sk}, m)$**:** *is a* PPT *algorithm that on input a secret key* sk *and message* $m \in \{0,1\}^*$*, outputs a signature* σ*.*

$\mathsf{Ver}(\mathsf{pk}, m, \sigma)$**:** *is a* DPT *algorithm that on input a public key* pk*, message* $m \in \{0,1\}^*$ *and signature* σ*, outputs a bit* b*.*

Every signature scheme must satisfy *correctness*, meaning that for every $\lambda \in \mathbb{N}$ and every message $m \in \{0,1\}^*$, we have that

$$\Pr\left[\mathsf{Ver}(\mathsf{pk}, m, \mathsf{Sig}(\mathsf{sk}, m)) = 1 \mid (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)\right] = 1.$$

The most common security requirement of a signature scheme is existential unforgeability under chosen message attack (EUF-CMA security for short). On high level, it guarantees a malicious party, that does not know the private key, cannot produce a valid signature on a message m even if he knows polynomially many valid signatures on messages of his choice (but different from m). We recall this notion in Definition 7.

Definition 7 (EUF-CMA Security)**.** *A signature scheme* Σ *is* EUF-CMA *secure if for every* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$\Pr[\mathsf{SigForge}_{\mathcal{A},\Sigma}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{SigForge}_{\mathcal{A},\Sigma}$ *is defined as follows:*

```
$\mathsf{SigForge}_{\mathcal{A},\Sigma}(\lambda)$
$\mathcal{Q} \leftarrow \emptyset$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot)}(\mathsf{pk})$
return $(m \notin \mathcal{Q} \wedge \mathsf{Ver}(\mathsf{pk}, m, \sigma))$
```

```
$\mathcal{O}_{\mathrm{S}}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

Existential unforgeability does not say anything about the difficulty of transforming a valid signature on m into another valid signature on m. Hardness of such transformation is captured by a stronger notion, called strong existential unforgeability under chosen message attack (or $\mathsf{SUF\text{-}CMA}$ for short), which we recall next.

Definition 8 ($\mathsf{SUF\text{-}CMA}$ Security)**.** *A signature scheme Σ is $\mathsf{SUF\text{-}CMA}$ secure if for every PPT adversary $\mathcal{A}$ there exists a negligible function negl such that*

$$\Pr[\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}$ is defined as follows:

```
$\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}(\lambda)$
    $\mathcal{Q} \leftarrow \emptyset$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
    $(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot)}(\mathsf{pk})$
    $\mathbf{return}\ ((m, \sigma) \notin \mathcal{Q} \wedge \mathsf{Ver}(\mathsf{pk}, m, \sigma))$
```

```
$\mathcal{O}_\mathrm{S}(m)$
    $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m, \sigma\}$
    $\mathbf{return}\ \sigma$
```

The advantage of the adversary $\mathcal{A}$ playing the game $\mathsf{StrongSigForge}$ is defined as follows:

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{StrongSigForge}} = \Pr[\mathsf{StrongSigForge}_{\mathcal{A},\Sigma}(\lambda) = 1]$$

Randomizable Blind Signature. In Section 6.2 we make use of a digital signature scheme that allows blind signing of a message and randomizing a signature. More precisely, given a commitment com (as defined in Section 2.2.7) to a message m, one blindly sign as $\sigma^* \leftarrow \mathsf{BlindSig}(\mathsf{sk}, \mathsf{com})$. Then, anyone with access to the decommitment information decom can unblind the signature σ^* to obtain a signature on m, i.e., $\sigma \leftarrow \mathsf{UnblindSig}(\mathsf{decom}, \sigma^*)$. Moreover, a signatures σ can be publicly randomized as $\sigma' \leftarrow \mathsf{RandSig}(\sigma)$. We call such a digital signatures scheme here as *randomizable blind signature (RBS)*, and denote it as Σ_{RBS}. An example of such a digital signature scheme is the Pointcheval-Sanders signature scheme [PS16, PS18].

We remark here that unlike the (classical) blind signature (BS) schemes that consider the one-more unforgeability and blindness notions [HKL19], in RBS the security notion is the aforementioned strong unforgeability notion, and the blindness comes from the hiding property of the underlying commitment scheme (see Section 2.2.7). Moreover, randomizable signatures can only achieve $\mathsf{EUF\text{-}CMA}$ security.

2.2.4 Two-Party Signature with Aggregatable Public Keys

We also make use of two-party signatures with aggregatable public keys, as defined by Erwig et al. [EFH+21], which naturally extends the digital signature definition given in Section 2.2.3 to a two-party setting.

Definition 9 (Two-Party Signature with Aggregatable Public Keys [EFH$^+$21])**.** *A two-party signature scheme with aggregatable public keys is a tuple of protocols and algorithms* $\Sigma_2 = (\mathsf{Setup}, \mathsf{KeyGen}, \Pi_{\mathsf{Sig}}, \mathsf{KAgg}, \mathsf{Ver})$ *defined as follows:*

$\mathsf{Setup}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* 1^λ*, outputs public parameters* pp.

$\mathsf{KeyGen}(\mathsf{pp})$**:** *is a* PPT *algorithm that on input public parameters* pp*, outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$.

$\Pi_{\mathsf{Sig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$**:** *is an interactive* PPT *protocol that on input secret keys* sk_i *from party* P_i *with* $i \in \{0,1\}$*, and common values messages* $m \in \{0,1\}^*$ *and public keys* $\mathsf{pk}_0, \mathsf{pk}_1$*, outputs a signature* σ.

$\mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$**:** *is a* DPT *algorithm that on input two public keys* $\mathsf{pk}_0, \mathsf{pk}_1$*, outputs an aggregated public key* apk.

$\mathsf{Ver}(\mathsf{apk}, m, \sigma)$**:** *is a* DPT *algorithm that on input an aggregate public key* apk*, message* $m \in \{0,1\}^*$ *and signature* σ*, outputs a bit* b.

We can define *completeness* for Σ_2 in a natural way: a two-party signature scheme with aggregatable public keys Σ_2 satisfies completeness, if for all public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, key pair $(\mathsf{sk}, \mathsf{pk})\mathsf{KeyGen}(\mathsf{pp})$ and messages $m \in \{0,1\}^*$, the protocol $\Pi_{\mathsf{Sig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$ outputs a signature σ to both parties P_0, P_1, such that $\mathsf{Ver}(\mathsf{apk}, m, \sigma) = 1$, where $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$.

A two-party signature scheme with aggregatable public keys should satisfy *unforgeability*. At a high level, this property guarantees that if one of the two parties is malicious, this party is not able to produce a valid signature under the aggregated public key without the cooperation of the other party. We formalize this property via an experiment $\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}$, where $b \in \{0,1\}$ defines which of the two parties is corrupted.

Definition 10 (2-EUF-CMA Security)**.** *A two-party signature scheme with aggregatable public keys* Σ_2 *is* 2-EUF-CMA *secure if for every* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that, for* $b \in \{0,1\}$*,*

$$\Pr[\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}(\lambda) = 1] \le \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{SigForge}^b_{\mathcal{A},\Sigma_2}$ *is defined as follows:*

```
$\mathsf{SigForge}^{b}_{\mathcal{A},\Sigma_2}(\lambda)$
    $\mathcal{Q} \leftarrow \emptyset$
    $\mathsf{pp} \leftarrow \mathsf{Setup}(1^{\lambda})$
    $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
    $(\mathsf{sk}_{b}, \mathsf{pk}_{b}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
    $(m, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}^{b}_{\Pi_{\mathrm{S}}}(\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_{b}, \mathsf{pk}_{b})$
    $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
    $\textbf{return} \ (m \notin \mathcal{Q} \land \mathsf{Ver}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^{b}_{\Pi_{\mathrm{S}}}(m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    $\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
    $\textbf{return} \ \sigma$
```

2.2.5 Adaptor Signature

Next, we give a formal description of an adaptor signature and its properties. Adaptor signatures have been introduced by the cryptocurrency community to tie together the authorization of a transaction with leakage of a secret value. Due to its utility, adaptor signatures have been used in previous works for various applications like atomic swaps or payment channel networks [MMS$^+$19]. 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 that knows a certain secret, where the completion of the signature reveals the underlying secret.

More precisely, we define an adaptor signature scheme with respect to a standard signature scheme Σ and a hard relation R. In an adaptor signature scheme, for any statement $Y \in L$, 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 full valid signature on m if and only if the adaptor knows a witness for Y. Moreover, if such a valid signature is produced, it must be possible to extract the witness for Y given the pre-signature and the adapted signature. This is formalized as follows, where we take the message space $\mathcal{M}$ to be $\{0,1\}^*$.

Definition 11 (Adaptor Signature Scheme)**.** *An adaptor signature scheme w.r.t. a hard relation R and a signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sig}, \mathsf{Ver})$ consists of four algorithms $\Xi_{R,\Sigma} = (\mathsf{PreSig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ defined as:*

$\mathsf{PreSig}(\mathsf{sk}, m, Y)$**:** *is a* PPT *algorithm that on input a secret key* sk*, message $m \in \{0,1\}^*$ and statement $Y \in L$, outputs a pre-signature $\hat{\sigma}$.*

$\mathsf{PreVer}(\mathsf{pk}, m, Y, \hat{\sigma})$**:** *is a* DPT *algorithm that on input a public key* pk*, message $m \in \{0,1\}^*$, statement $Y \in L$ and pre-signature $\hat{\sigma}$, outputs a bit b.*

$\mathsf{Adapt}(\hat{\sigma}, y)$**:** *is a* DPT *algorithm that on input a pre-signature $\hat{\sigma}$ and witness y, outputs a signature σ.*

$\mathsf{Ext}(\sigma, \hat{\sigma}, Y)$**:** *is a* DPT *algorithm that on input a signature σ, pre-signature $\hat{\sigma}$ and statement $Y \in L$, outputs a witness y such that $(Y, y) \in R$, or $\perp$.*

We note that an adaptor signature scheme $\Xi_{R,\Sigma}$ also inherits the KeyGen and Ver algorithms from the underlying signature scheme Σ. In addition to the standard signature correctness, an adaptor signature scheme has to satisfy *pre-signature correctness*. Informally, an honestly generated pre-signature w.r.t. a statement $Y \in L$ is a valid pre-signature and can be adapted into a valid signature from which a witness for Y can be extracted.

Definition 12 (Pre-signature Correctness)**.** *An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies pre-signature correctness if for every $\lambda \in \mathbb{N}$, every message $m \in \{0,1\}^*$ and every statement/witness pair $(Y, y) \in R$, the following holds:*

$$\Pr\left[\begin{array}{c} \mathsf{PreVer}(\mathsf{pk}, m, Y, \hat{\sigma}) = 1 \\ \wedge \\ \mathsf{Ver}(\mathsf{pk}, m, \sigma) = 1 \\ \wedge \\ (Y, y') \in R \end{array} \middle| \begin{array}{l} (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ \hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, Y) \\ \sigma := \mathsf{Adapt}(\hat{\sigma}, y) \\ y' := \mathsf{Ext}(\sigma, \hat{\sigma}, Y) \end{array}\right] = 1.$$

Next, we define the security properties of an adaptor signature scheme. We start with the notion of unforgeability, which is similar to existential unforgeability under chosen message attacks ($\mathsf{EUF\text{-}CMA}$) 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 $Y \in L$. We note that allowing the adversary to learn a pre-signature on the forgery message m is crucial as for our applications unforgeability needs to hold even in case the adversary learns a pre-signature for m without knowing a witness for Y. We now formally define the existential unforgeability under chosen message attack for adaptor signature ($\mathsf{aEUF\text{-}CMA}$).

Definition 13 ($\mathsf{aEUF\text{-}CMA}$ Security)**.** *An adaptor signature scheme $\Xi_{R,\Sigma}$ is $\mathsf{aEUF\text{-}CMA}$ secure if for every PPT adversary $\mathcal{A}$ there exists a negligible function negl such that:* $\Pr[\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$*, where the experiment $\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}$ is defined as follows:*

```
$\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$
    $\mathcal{Q} := \emptyset$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
    $m \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
    $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, Y)$
    $\sigma \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\hat{\sigma}, Y)$
    return $(m \notin \mathcal{Q} \wedge \mathsf{Ver}(\mathsf{pk}, m, \sigma))$

$\mathcal{O}_\mathrm{S}(m)$
    $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\sigma$

$\mathcal{O}_\mathrm{pS}(m, Y)$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, Y)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\hat{\sigma}$
```

An additional property that we require from adaptor signatures is *pre-signature adaptability*, which states that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be adapted into a valid signature using the witness y with $(Y, y) \in R$. We note that this property is stronger than the pre-signature correctness property from Definition 12, since we require that even maliciously produced pre-signatures can always be completed into valid signatures. The following definition formalizes this property.

Definition 14 (Pre-signature Adaptability). *An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies pre-signature adaptability if for any $\lambda \in \mathbb{N}$, any message $m \in \{0,1\}^*$, any statement/witness pair $(Y, y) \in R$, any key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and any pre-signature $\hat{\sigma} \leftarrow \{0,1\}^*$ with $\mathsf{PreVer}(\mathsf{pk}, m, Y, \hat{\sigma}) = 1$, we have*

$$\Pr[\mathsf{Ver}(\mathsf{pk}, m, \mathsf{Adapt}(\hat{\sigma}, y)) = 1] = 1.$$

The last property that we are interested in is *witness extractability*. Informally, it guarantees that a valid signature/pre-signature pair $(\sigma, \hat{\sigma})$ for a message/statement pair (m, Y) can be used to extract the corresponding witness y of Y.

Definition 15 (Witness Extractability). *An adaptor signature scheme $\Xi_{R,\Sigma}$ is* witness extractable *if for every* PPT *adversary $\mathcal{A}$, there exists a negligible function* negl *such that the following holds:* $\Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$*, where the experiment* $\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}$ *is defined as follows*

```
$\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(\lambda)$
    $\mathcal{Q} := \emptyset$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
    $(m, Y) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, Y)$
    $\sigma \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\hat{\sigma})$
    $y' := \mathsf{Ext}(\mathsf{pk}, \sigma, \hat{\sigma}, Y)$
    $\mathbf{return}\ (m \notin \mathcal{Q} \wedge (Y, y') \notin R \wedge \mathsf{Ver}(\mathsf{pk}, m, \sigma))$
```

```
$\mathcal{O}_{\mathrm{S}}(m)$
    $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    $\mathbf{return}\ \sigma$
```

```
$\mathcal{O}_{\mathrm{pS}}(m, Y)$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, Y)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    $\mathbf{return}\ \hat{\sigma}$
```

Although, the witness extractability experiment aWitExt looks similar to the experiment aSigForge, there is one important difference, namely, the adversary is allowed to choose the forgery statement Y. Hence, we can assume that the adversary knows a witness for Y, and therefore, 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.

Combining the three properties described above, we can define a secure adaptor signature scheme as follows.

Definition 16 (Secure Adaptor Signature Scheme)**.** *An adaptor signature scheme* $\Xi_{R,\Sigma}$ *is secure, if it is* $\mathsf{aEUF\text{-}CMA}$ *secure, pre-signature adaptable and witness extractable.*

2.2.6 Two-Party Adaptor Signature with Aggregatable Public Keys

In this section, we describe two-party adaptor signature with aggregatable public keys, which is a natural extension of adaptor signatures given in Section 2.2.5 to a two-party setting.

Definition 17 (Two-Party Adaptor Signature Scheme with Aggregatable Public Keys [EFH+21])**.** *A two-party adaptor signature scheme with aggregatable public keys is defined w.r.t. a hard relation* R *and a two-party signature scheme with aggregatable public keys* $\Sigma_2 = (\mathsf{Setup}, \mathsf{KeyGen}, \Pi_{\mathsf{Sig}}, \mathsf{KAgg}, \mathsf{Ver})$*. It is run between parties* P_0, P_1 *and consists of a tuple* $\Xi_{R,\Sigma_2} = (\Pi_{\mathsf{Pre}}\mathsf{Sig}, \mathsf{Adapt}, \mathsf{PreVer}, \mathsf{Ext})$ *of efficient protocols and algorithms defined as follows:*

$\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$**:** *is an interactive protocol with input secret key* sk_i *from party* P_i *with* $i \in \{0,1\}$ *and common message* $m \in \{0,1\}^*$*, public keys* $\mathsf{pk}_0, \mathsf{pk}_1$ *and statement* $Y \in L_R$*, outputs a pre-signature* $\hat{\sigma}$*.*

$\mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma})$**:** *is a* DPT *algorithm with input an aggregated public key* apk*, a message* $m \in \{0,1\}^*$*, a statement* $Y \in L_R$ *and a pre-signature* $\hat{\sigma}$*, outputs bit* b*.*

$\mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y)$**:** *is a* DPT *algorithm with input an aggregated public key* apk*, pre-signature* $\hat{\sigma}$ *and witness* y*, outputs a signature* σ*.*

$\mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$**:** *is a* DPT *algorithm with input an aggregated public key* apk*, a signature* σ*, pre-signature* $\hat{\sigma}$ *and statement* $Y \in L_R$*, outputs a witness* y *s.t.* $(Y, y) \in R$*, or* $\perp$*.*

We first define the two-party pre-signature correctness.

Definition 18 (Two-Party Pre-signature Correctness)**.** *A two-party adaptor signature scheme with aggregatable public keys* Ξ_{R,Σ_2} *satisfies two-party pre-signature correctness if for every* $\lambda \in \mathbb{N}$*, every message* $m \in \{0,1\}^*$ *and every statement/witness pair* $(Y, y) \in R$*, the following holds:*

$$\Pr\left[\begin{array}{c|l} & \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma}) = 1 & (\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KeyGen}(\mathsf{pp}) \\ \wedge & (\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{KeyGen}(\mathsf{pp}) \\ \mathsf{Ver}(\mathsf{apk}, m, \sigma) = 1 & (Y, y) \leftarrow \mathsf{GenR}(1^\lambda) \\ \wedge & \hat{\sigma} \leftarrow \Pi_{\mathsf{PreSig}\langle \mathsf{sk}_0, \mathsf{sk}_1\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y) \\ (Y, y') \in R & \mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1) \\ & \sigma := \mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y) \\ & y' := \mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y) \end{array}\right] = 1.$$

We now formally define the existential unforgeability under chosen message attack for two-party adaptor signature scheme with aggregatable public keys (2-$\mathsf{aEUF\text{-}CMA}$). It is similar to the two-party existential unforgeability under chosen message attacks (given in Definition 10) 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 $Y \in L_R$.

Definition 19 (2-aEUF-CMA Security)**.** *A two-party adaptor signature scheme with aggregatable public keys* Ξ_{R,Σ_2} *is* 2-aEUF-CMA *secure if for every* PPT *adversary* $\mathcal{A}$ *there exists a negligible function* negl *such that.*

$$\Pr[\mathsf{aSigForge}^{b}_{\mathcal{A},\Xi_{R,\Sigma_2}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{aSigForge}^{b}_{\mathcal{A},\Xi_{R,\Sigma_2}}$ *is defined as follows:*

```
$\mathsf{aSigForge}^{b}_{\mathcal{A},\Xi_{R,\Sigma_2}}(\lambda)$
    $\mathcal{Q} := \emptyset$
    $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
    $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
    $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
    $m \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_b, \mathsf{pk}_b)$
    $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$
    $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(m, Y)$
    $\sigma \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathrm{pS}}(\cdot,\cdot)}(\hat{\sigma}, Y)$
    $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
    return $(m \notin \mathcal{Q} \wedge \mathsf{Ver}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^b_{\Pi_\mathrm{S}}(m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    $\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
    return $\sigma$

$\mathcal{O}^b_{\Pi_\mathrm{pS}}(m, Y)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
    return $\hat{\sigma}$
```

Next, we formally define the property of *pre-signature adaptability* for two-party adaptor signature with aggregatable keys.

Definition 20 (Two-Party Pre-signature Adaptability)**.** *A two-party adaptor signature scheme with aggregatable public keys* Ξ_{R,Σ_2} *satisfies two-party pre-signature adaptability if for any* $\lambda \in \mathbb{N}$*, any message* $m \in \{0,1\}^*$*, any statement/witness pair* $(Y, y) \in R$*, any public keys* pk_0 *and* pk_1*, and any pre-signature* $\hat{\sigma} \leftarrow \{0,1\}^*$ *satisfying* $\mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma}) = 1$*, where* $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$*, we have*

$$\Pr[\mathsf{Ver}(\mathsf{apk}, m, \mathsf{Adapt}(\mathsf{apk}, \hat{\sigma}, y)) = 1] = 1.$$

The last property that we are interested in is *witness extractability*, which in two-party setting can be defined as a natural adaptation of the definition given in Definition 15.

Definition 21 (Two-Party Witness Extractability)**.** *A two-party adaptor signature scheme with aggregatable public keys* Ξ_{R,Σ_2} *two-party witness extractable if for every* PPT *adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that the following holds:*

$$\Pr[\mathsf{aWitExt}^{b}_{\mathcal{A},\Xi_{R,\Sigma_2}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{aWitExt}^b_{\mathcal{A},\Xi_{R,\Sigma_2}}$ *is defined as follows*

```
$\mathsf{aWitExt}^b_{\mathcal{A},\Xi_{R,\Sigma_2}}(\lambda)$
  $\mathcal{Q} := \emptyset;\ \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
  $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
  $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}_{1-b})$
  $(m, Y) \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk}_{1-b}, \mathsf{sk}_b, \mathsf{pk}_b)$
  $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(m, Y)$
  $\sigma \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}(\cdot), \mathcal{O}^b_{\Pi_\mathrm{pS}}(\cdot,\cdot)}(\hat{\sigma})$
  $\mathsf{apk} := \mathsf{KAgg}(\mathsf{pk}_0, \mathsf{pk}_1)$
  $y' := \mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$
  return $(m \notin \mathcal{Q} \land (Y, y') \notin R \land \mathsf{Ver}(\mathsf{apk}, m, \sigma))$

$\mathcal{O}^b_{\Pi_\mathrm{S}}(m)$
  $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
  $\sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m)$
  return $\sigma$

$\mathcal{O}^b_{\Pi_\mathrm{pS}}(m, Y)$
  $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
  $\hat{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
  return $\hat{\sigma}$
```

Combining the three properties described above, we can define a secure two-party adaptor signature scheme with aggregatable public keys as follows.

Definition 22 (Secure Two-Party Adaptor Signature Scheme with Aggregatable Public Keys)**.** *A two-party adaptor signature scheme with aggregatable public keys* Ξ_{R,Σ_2} *is secure, if it is* 2-$\mathsf{aEUF\text{-}CMA}$ *secure, two-party pre-signature adaptable and two-party witness extractable.*

Generic Transformation. Erwig et al. [EFH+21] showed how to generically transform a canonical identification scheme (as defined in Section 2.2.2) into an (two-party) adaptor signature scheme. Here we describe the generic transformation from two-party signature with aggregatable public keys (obtained from an identification scheme) to an adaptor signature scheme, given in [EFH+21, Section 5.1]. This transformation is given in Figure 2.1, and it makes use of the following functions and protocols:

- The randomness shift function $f_{\mathsf{shift}} \colon \mathcal{D}_{\mathsf{rand}} \times L \rightarrow \mathcal{D}_{\mathsf{rand}}$, takes as input a commitment value $R \in \mathcal{D}_{\mathsf{rand}}$ of the identification scheme and a statement $Y \in L$ of the hard relation, and outputs a new commitment value $R' \in \mathcal{D}_{\mathsf{rand}}$. We assume that the inverse of this function is well-defined (which is true for all currently known instantiations of it).
- The adaptation function $f_{\mathsf{adapt}} \colon \mathcal{D}_{\mathsf{resp}} \times \mathcal{D}_{\mathsf{w}} \rightarrow \mathcal{D}_{\mathsf{resp}}$, takes as input a pre-signature value $\hat{s} \in \mathcal{D}_{\mathsf{resp}}$ (which corresponds to the response value of the identification scheme) and a witness $y \in \mathcal{D}_{\mathsf{w}}$ of the hard relation R, and outputs a new value $s \in \mathcal{D}_{\mathsf{resp}}$.
- The witness extraction function $f_{\mathsf{ext}} \colon \mathcal{D}_{\mathsf{resp}} \times \mathcal{D}_{\mathsf{resp}} \rightarrow \mathcal{D}_{\mathsf{w}}$, takes as input two response values $\hat{s}, s \in \mathcal{D}_{\mathsf{resp}}$ and outputs a witness $y \in \mathcal{D}_{\mathsf{w}}$.
- The randomness combining function $f_{\mathsf{com\text{-}rand}} \colon \mathcal{D}_{\mathsf{rand}} \times \mathcal{D}_{\mathsf{rand}} \rightarrow \mathcal{D}_{\mathsf{rand}}$, that takes as input two randomness $R_0, R_1 \in \mathcal{D}_{\mathsf{rand}}$ and outputs a new combined randomness $R \in \mathcal{D}_{\mathsf{rand}}$.

```
$\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_i, \mathsf{sk}_{1-i}\rangle}(\mathsf{pk}_0, \mathsf{pk}_1, m, Y)$
    Parse $\mathsf{pk}_i = ((1^\lambda, \mathsf{pp}_C, \mathsf{crs}), \mathsf{pk}'_i), i \in \{0,1\}$
    $(R_i, \mathsf{st}_i, R_{1-i}) \leftarrow \Pi_{\mathsf{Rand\text{-}Exc}\langle sk_i, sk_{1-i}\rangle}(\mathsf{pp}_C, \mathsf{crs})$
    $R_{\mathsf{pre}} := f_{\mathsf{com\text{-}rand}}(R_0, R_1)$
    $R_{\mathsf{sign}} := f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y), \ h := H(R_{\mathsf{sign}}, m)$
    $\hat{s}_i \leftarrow \mathsf{P}_2(\mathsf{sk}_i, R_i, h, \mathsf{st}_i)$
    $\hat{s}_{i-1} \leftarrow \Pi_{\mathsf{Exchange}}\langle \hat{s}_i, \hat{s}_{i-1}\rangle$
    $(h, \hat{s}) := f_{\mathsf{com\text{-}sig}}(h, (\hat{s}_i, \hat{s}_{i-1}))$
    return $\hat{\sigma} := (h, \hat{s})$

$\mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma} := (h, \hat{s}))$
    $\widehat{R}_{\mathsf{pre}} := \mathsf{V}_0(\mathsf{apk}, h, \hat{s})$
    return $h = H(f_{\mathsf{shift}}(\widehat{R}_{\mathsf{pre}}, Y), m)$

$\mathsf{Adapt}(\mathsf{apk}, \hat{\sigma} := (h, \hat{s}), y)$
    return $\sigma := (h, f_{\mathsf{adapt}}(\hat{s}, y))$

$\mathsf{Ext}(\mathsf{apk}, \sigma := (h, s), \hat{\sigma} := (h, \hat{s}), Y)$
    return $f_{\mathsf{ext}}(s, \hat{s})$
```

Figure 2.1: Two-party adaptor signature scheme with aggregatable public keys Ξ_{R,Σ_2} with respect to Σ_2 and hard relation R.

- The signature combining function $f_{\mathsf{com\text{-}sig}}$, that takes as input two partial signatures and returns a new combined signature.
- The randomness exchange protocol $\Pi_{\mathsf{Rand\text{-}Exc}}$.
- The partial signature exchange protocol Π_{Exchange}.

In [EFH$^+$21] it was shown how to instantiate the functions and protocols specified above for different type of signatures, such as Schnorr [Sch91], Katz-Wang [KW03] and Guillou-Quisquater [GQ88]. We note here that the recently proposed post-quantum adaptor signatures, such as lattice-based LAS [EEE20] and isogeny-based IAS [TMM21b] are also adaptor signature scheme obtained from an identification scheme. However, we note that LAS [EEE20] can only achieve weak pre-signature adaptability due to the inherent knowledge gap in lattice-based schemes, and IAS [TMM21b] cannot be extended to two-party setting as no known key aggregation technique exists in isogeny-based cryptography.

2.2.7 Commitment Scheme

A commitment scheme allows a user to commit to a message by generating a commitment that can only be opened given the corresponding opening information. We formally define a commitment scheme as follows:

Definition 23 (Commitment Scheme)**.** *A (non-interactive) commitment scheme consists of pair of algorithms* $\Pi_{\mathsf{COM}} = (\mathsf{Commit}, \mathsf{Open})$ *defined as follows:*

$\mathsf{Commit}(1^\lambda, m)$**:** *is a* PPT *algorithm that on input a security parameter* 1^λ *and a message* $m \in \{0,1\}^*$*, outputs a commitment* com *and an opening information* decom*.*

$\mathsf{Open}(\mathsf{com}, \mathsf{decom})$**:** *is a* DPT *algorithm that on input a commitment* com *and an opening information* decom*, outputs either* m *or* $\perp$*.*

We require the standard notion of *correctness*, which says that for every $\lambda \in \mathbb{N}$ and every message $m \in \{0,1\}^*$, it holds that

$$\Pr\left[\mathsf{Open}(\mathsf{Commit}(1^\lambda, m)) = m\right] = 1.$$

In terms of security we require the commitment scheme to satisfy *computational hiding* and *perfect binding* properties [Dam99].

Definition 24 (Computational Hiding)**.** *A commitment scheme* Π_{COM} *is computationally hiding if for every* PPT *adversary* $\mathcal{A}$*, there exists a negligible function* negl*, such that*

$$\Pr\left[\mathsf{ComHiding}_{\mathcal{A},\Pi_{\mathsf{COM}}}(\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{ComHiding}_{\mathcal{A},\Pi_{\mathsf{COM}}}$ *is defined as follows*

```
$\mathsf{ComHiding}_{\mathcal{A},\Pi_{\mathsf{COM}}}(\lambda)$

$(m_0, m_1) \leftarrow \mathcal{A}(1^\lambda)$
$b \leftarrow\!\$\ \{0,1\}$
$(\mathsf{com}, \mathsf{decom}) \leftarrow \mathsf{Commit}(1^\lambda, m_b)$
$b' \leftarrow \mathcal{A}(\mathsf{com})$
return $b = b'$
```

Definition 25 (Perfect Binding)**.** *A commitment scheme* Π_{COM} *is perfectly binding if for every* $\lambda \in \mathbb{N}$*, no (computationally unbounded) adversary* $\mathcal{A}$ *can output a tuple* $(\mathsf{com}, \mathsf{decom}, \mathsf{decom}')$*, such that* $m \leftarrow \mathsf{Open}(\mathsf{com}, \mathsf{decom})$*,* $m' \leftarrow \mathsf{Open}(\mathsf{com}, \mathsf{decom}')$ *and* $m \neq m' \neq bot$*.*

2.2.8 Encryption Scheme

In this section we give a formal description of a (public-key) encryption scheme and its properties.

Definition 26 (Encryption Scheme)**.** *An encryption scheme* $\Pi_{\mathsf{E}} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *with message space* $\mathcal{M}$ *consists of the following algorithms:*

$\mathsf{KeyGen}(1^\lambda)$**:** *is a* PPT *algorithm that on input a security parameter* 1^λ*, outputs an encryption/decryption key pair* $(\mathsf{ek}, \mathsf{dk})$*.*

$\mathsf{Enc}(\mathsf{ek}, m)$**:** *on input an encryption key* ek *and a message* m*, outputs a ciphertext* c*.*

$\mathsf{Dec}(\mathsf{dk}, c)$**:** *on input a decryption key* dk *and a ciphertext* c*, outputs a message* $m \in \mathcal{M} \cup \{\perp\}$

We say that an encryption scheme Π_{E} is *perfectly correct* if for all $\lambda \in \mathbb{N}$, for all $(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ and for all $m \in \mathcal{M}$ it holds that,

$$\Pr\left[\mathsf{Dec}(\mathsf{dk}, \mathsf{Enc}(\mathsf{ek}, m)) = m\right] = 1.$$

Next, we recall the standard notion of indistinguishability under chosen ciphertext attacks (IND-CCA security).

Definition 27 (IND-CCA)**.** *An encryption scheme Π_{E} is* IND-CCA *secure if for every* PPT *adversary $\mathcal{A}$, there exists a negligible function* negl, *such that*

$$\Pr\left[\mathsf{IND\text{-}CCA}_{\mathcal{A},\Pi_{\mathsf{E}}}(\lambda) = 1\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where the experiment $\mathsf{IND\text{-}CCA}_{\mathcal{A},\Pi_{\mathsf{E}}}$ is defined as follows,

```
$\mathsf{IND\text{-}CCA}_{\mathcal{A},\Pi_{\mathsf{E}}}(\lambda)$
$(\mathsf{ek}, \mathsf{dk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m_0, m_1) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Dec}}(\cdot)}(\mathsf{ek})$
$b \leftarrow\$ \{0,1\}$
$c \leftarrow \mathsf{Enc}(\mathsf{ek}, m_b)$
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Dec}}(\cdot)}(c)$
return $b = b'$
```

for a decryption oracle $\mathcal{O}_{\mathsf{Dec}}(\cdot)$ that outputs $\perp$ if queried on the challenge ciphertext c.

In Section 6.2 we consider an encryption scheme that has unique decryption keys, which we formalize below.

Definition 28 (Unique Decryption Keys)**.** *An encryption scheme Π_{E} has unique decryption keys if there exists an algorithm* Gen, *such that* KeyGen *algorithm is of the following form:*

- *Sample* $\mathsf{dk} \leftarrow\$ \{0,1\}^\lambda$.
- *Compute* $\mathsf{ek} \leftarrow \mathsf{Gen}(\mathsf{dk})$.
- *Output* $(\mathsf{ek}, \mathsf{dk})$.

Moreover, for all ek *output by* KeyGen, *there exists a unique* dk, *such that* $\mathsf{ek} \leftarrow \mathsf{Gen}(\mathsf{dk})$ *holds, i.e.,* Gen *is injective.*

We note that this property is already satisfied by most natural public-key encryption schemes, but it can be generically achieved by augmenting the encryption key ek with a perfectly binding commitment to the decryption key dk, i.e., $\mathsf{com} \leftarrow \Pi_{\mathsf{COM}}.\mathsf{Commit}(1^\lambda, \mathsf{dk})$.

2.3 Security and Communication Model

In this section we introduce the adversarial and communication model that we consider for our protocols.

2.3.1 Universal Composability

To model security of our protocols in the presence of concurrent executions of different protocols and allow for universal composition of building blocks, we resort to the *Universal Composability (UC)* framework of Canetti [Can20] and its extended variant that supports a global setup, called *Universal Composition with Global Subroutines (UCGS)* [BCH$^+$20]. The following introduction is primary adapted from [Can20] and [Hos21].

An n-party protocol π is a collection of n different programs, where each is formally modeled as an interactive Turing machine (ITM), designed to reach a joint and specific goal. Each program is expected to be executed by a different computational entity while exchanging messages with the remaining programs from the collection. In this thesis, we often abstract from the distinction between an ITM and the entity operating it, and refer to both as a *party* in the protocol π. Moreover, we assume that the set $\mathcal{P} = \{P_1, \dots, P_n\}$ of parties using the system is fixed.

A protocol π is executed in the presence of two additional entities, called the *environment* $\mathcal{E}$ and the *adversary* $\mathcal{A}$, both of which are modeled as ITMs. The role of the environment is to present any external information to the current protocol execution. The environment $\mathcal{E}$ provides input to both the adversary $\mathcal{A}$ and the parties $\mathcal{P}$ and observe their outputs. The adversary $\mathcal{A}$ can *corrupt* any party in $\mathcal{P}$ at the beginning of the protocol execution, i.e., we assume *static* corruption. When a party P is corrupted, the adversary $\mathcal{A}$ takes full control over a P's actions and learns its internal state. The output of an environment $\mathcal{E}$ interacting with a protocol π and an adversary $\mathcal{A}$ on input the security parameter 1^λ and auxiliary input z is denoted by $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(\lambda, z)$.

An important concept in the UC framework is *protocol emulation.* At a high level, a protocol π *emulates* another protocol ϕ if no PPT environment $\mathcal{E}$ is able to distinguish whether it is interacting with the protocol π executed in the presence of an adversary of its choice, or the protocol ϕ in the presence of another adversary, referred to as the *simulator.* The formal definition of UC emulation is as follows.

Definition 29 (UC Emulation [Can20])**.** *The protocol π UC-emulates the protocol ϕ, if for every* PPT *adversary $\mathcal{A}$ there exists a* PPT *adversary $\mathcal{S}$ such that for every* PPT *environment $\mathcal{E}$ we have*

$$\{\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}} \approx_c \{\mathsf{EXEC}_{\phi,\mathcal{S},\mathcal{E}}(\lambda, z)\}_{\substack{\lambda\in\mathbb{N},\\ z\in\{0,1\}^*}}$$

(where $\approx_c$ denotes computational indistinguishability).

In order to formalize the security of protocols, the UC framework utilizes the concept of *ideal functionalities.* An ideal functionality is a single ITM defining the ideal behavior of the protocol we aim to design. This means that it defines the input/output behavior of parties, and it specifies what information may be disclosed to the adversary during the protocol execution. An *ideal protocol* defining the tasks for n parties consists of an ideal functionality $\mathcal{F}$ and n *dummy parties*. These dummy parties are special ITMs that upon receiving an input, directly forward it to the ideal functionality $\mathcal{F}$ together with the information about the sender of the message. If dummy parties receive an output from $\mathcal{F}$ addressed to a certain entity, the dummy parties simply forward this output to the specified destination. We denote the ideal protocol for an ideal functionality $\mathcal{F}$ as $\phi_{\mathcal{F}}$, and formalize UC-realization as follows.

Definition 30 (UC Realization [Can20]). *The protocol π UC-realizes the ideal functionality $\mathcal{F}$ if it UC-emulates the ideal protocol $\phi_{\mathcal{F}}$.*

We refer to the execution of the ideal protocol $\phi_{\mathcal{F}}$ in the presence of a simulator $\mathcal{S}$ as the *ideal world*, and to the execution of the real protocol π in the presence of an adversary $\mathcal{A}$ as the *real world*.

The main advantage of the UC framework is its support for secure composition of protocols. In order to state the UC composition theorem, we need to introduce several technical terms first (we refer to [Can20] for the formal definitions).

Parties of a protocol π may have access to one or multiple *subroutines*. These are ITMs that can communicate only with parties from the protocol π or with each other, e.g., they do not communicate with the environment $\mathcal{E}$. Formally, subroutine ITMs are part of the collection forming π and it must hold that (i) if a party P_i makes calls to a subroutine ITM, then this ITM is part of π, and (ii) if an ITM which is part of π accepts subroutine calls from a party P_i, then P_i is a party of the protocol π. We say that ϕ is a *subroutine protocol* of the protocol π if: (i) ϕ is a valid protocol itself, and (ii) ϕ consists of ITMs that are all subroutines of π. In the special case when ϕ is the ideal protocol of some functionality $\mathcal{G}$, we say that π works in the *$\mathcal{G}$-hybrid world*. A protocol π can have multiple subroutine protocols $\phi_1, \dots, \phi_m$. We note that since ITMs in each ϕ_i can communicate only with each other and with parties of the protocol π, no iteration between the subroutine protocols is possible. Two protocols ϕ and ρ are called *identity-compatible* if there is an injective correspondence between the parties of the protocol ϕ and the protocol ρ. Furthermore, the external ITMs that are allowed to interact with parties from ϕ and ρ are the same.

Now consider three protocols π, ϕ and ρ, where ϕ is subroutine protocol of π, and ϕ and ρ are identity-compatible. The *composed protocol*, denoted as $\pi^{\phi \to \rho}$, is defined exactly as π but replaces all calls to ϕ with calls to ρ. If ϕ is an ideal protocol of some ideal functionality $\mathcal{G}$, we write $\pi^{\mathcal{G} \to \rho}$ instead of $\pi^{\phi_{\mathcal{G}} \to \rho}$. We are not ready to state the UC composition theorem, which says that if ρ UC-emulates ϕ, then the composed protocol UC emulates the original protocol π. In other words, no PPT environment can distinguish between the interaction with the composed protocol and the original one.

Definition 31 (Universal Composability [Can20])**.** *Let π, ϕ and ρ be three protocols, where ϕ is a subroutine protocol of π, ϕ and ρ are identity-compatible, and ρ UC-emulates ϕ. Then, $\pi^{\phi \to \rho}$ UC-emulates π.*

A useful implication of the UC composition theorem is that if the protocol π working in the $\mathcal{G}$-hybrid world UC-realizes an ideal functionality $\mathcal{F}$, and we know that a protocol ρ UC realizes the ideal functionality $\mathcal{G}$, then the composed protocol $\pi^{\mathcal{G} \to \rho}$ also realizes the ideal functionality $\mathcal{F}$. This allows us to design a protocol realizing $\mathcal{F}$ in several steps. More precisely, we can first show how to design a protocol realizing the ideal functionality while using some ideal building blocks, and then we can show how to realize these ideal building blocks.

Global Ideal Functionalities. A disadvantage of the UC framework described above is that it does not allow for a protocol π to share state with another protocol π'. Therefore, it does not capture any *global setup* known to several protocol sessions and different protocols. In the context of cryptocurrencies, this is particularly problematic since we are not able to formally capture the public nature of the blockchain. Note that in practice, the blockchain can be read and even modified by multiple protocols in parallel which is impossible to express in the standard UC model.

Towards this end we utilize the UCGS framework introduced by Badertscher et al. [BCH$^+$20], which allows protocols to share state but in a controlled way. This is formalized using *global functionalities*. A global functionality is defined exactly as a standard ideal functionality except that it can communicate with more than one protocol session.

In order to capture global functionalities within the standard UC framework, Badertscher et al. [BCH$^+$20] defined a so-called *management protocol* $\mathsf{M}[\cdot]$, which is a transformation that takes two protocols π and γ, and combines them into a single protocol $\mu := \mathsf{M}[\pi, \gamma]$, such that one instance of μ behaves like one instance of π and one or more instances of γ, where the instances of γ take inputs both from the instance of π within μ, and from outside μ. In the rest of this description we can consider that $\gamma := \phi_{\mathcal{G}}$, i.e., the ideal protocol of some global functionality $\mathcal{G}$. Using such a management protocol we can define the UC-emulation with global subroutines as follows.

Definition 32 (UC Emulation with Global Subroutines [BCH$^+$20])**.** *Let π, ϕ and γ be protocols. We say that π UC-emulates ϕ in the presence of γ if protocol $\mathsf{M}[\pi, \gamma]$ UC-emulates protocol $\mathsf{M}[\phi, \gamma]$.*

It was observed in [BCH$^+$20] that in order to use global subroutines within UC, the protocols should be compliant, subroutine-exposing, subroutine respecting and regular (we refer to [BCH$^+$20] for formal definitions). Compliance and subroutine-exposing properties ensure that the protocols are identity-compatible and communication between different ITM instances (ITIs) is trustworthy. A protocol π is called γ-*subroutine respecting* if γ is the only global setup π shares state with. More precisely, ITIs of the current session of π communicate only with each other, their subroutines and instances of the global setup

γ. On the other hand, the regularity condition prevents the shared subroutine itself to spawn new higher-level sessions. More precisely, we say that γ is a ϕ-regular setup if, in any execution, the main parties of an instance of γ do not invoke a new ITI of ϕ. In general, most of the global setups used in the literature satisfy these restrictions. For example, a global clock only tells the time on demand, or a global ledger requires parties to register before participating in the protocol.

Badertscher et al. [BCH+20] showed that if a protocol ρ is γ-subroutine respecting, where γ itself is ρ-regular and subroutine respecting, then the interaction between π and the global subroutine γ is very structured without unexpected artifacts.

Now consider again three protocols π, ϕ and ρ, where ϕ is subroutine protocol of π. The universal composition with global subroutines is the composition theorem for protocols that are defined with respect to a global subroutine γ. Note that γ is not replaced, but ϕ is replaced by its implementation ρ.

Definition 33 (Universal Composition with Global Subroutines [BCH+20])**.** *Let π, ϕ, ρ and γ be subroutine-exposing protocols, where γ is a ϕ-regular setup and subroutine respecting, ϕ, ρ are γ-subroutine respecting and π is (ρ, ϕ)-compliant and $(\rho, M[\mathsf{code}, \gamma])$-compliant for $\mathsf{code} \in \{\phi, \rho\}$. If ρ UC-emulates ϕ in the presence of γ, then $\pi^{\phi \to \rho}$ UC-emulates π.*

Throughout the rest of this thesis we will denote the global functionalities with $\mathcal{G}$.

2.3.2 Synchrony and Communication

In all protocols presented in this thesis, we assume synchronous communication between parties. This means that the protocol execution happens in rounds and this is kept track using the global clock functionality $\mathcal{G}_{\mathsf{Clock}}$ [KMTZ13, BCH+20], depicted in Figure 2.2.

The global clock functionality $\mathcal{G}_{\mathsf{Clock}}$ mimics a clock by waiting for all honest parties and ideal functionalities to indicate that they are ready to proceed to the next round, and only then proceeds the clock to the next round. Moreover, the functionality also informs all the registered parties and functionalities of the given round. In practice such a clock synchrony among the parties and protocols can be achieved over the Internet.

2.4 Blockchain and Payment Channels

Blockchain. A *blockchain* constitutes a sequence of blocks containing some data, which achieves *immutability*, i.e., the data of the added blocks cannot be tampered with, and *liveness*, i.e., a new block is appended to the blockchain in regular time intervals. The blockchain is maintained over a decentralized network and a set of parties run consensus mechanisms to make a collective decision on the inclusion of new blocks. Some examples of such consensus mechanisms include Proof-of-Work (used in Bitcoin and formalized by Garay et al. [GKL15]) and Proof-of-Stake (used in Ethereum and formalized

Ideal Functionality $\mathcal{G}_{\mathsf{Clock}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $P := (\mathsf{pid}, \mathsf{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} = \emptyset$ and $F = \emptyset$.
For each session sid the clock maintains a variable τ_{sid}. For each identity $P := (\mathsf{pid}, \mathsf{sid}) \in \mathcal{P}$ it manages variable d_P. For each pair $(\mathcal{F}, \mathsf{sid}) \in F$ it manages variable $d_{\mathcal{F},\mathsf{sid}}$ (all integer variables are initially 0).

Synchronization:

- Upon receiving $(\mathsf{clock\text{-}update}, \mathsf{sid}_C)$ from some party $P \in \mathcal{P}$ set $d_P = 1$, execute **Round-Update** and forward $(\mathsf{clock\text{-}update}, \mathsf{sid}_C, P)$ to $\mathcal{S}$.
- Upon receiving $(\mathsf{clock\text{-}update}, \mathsf{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \mathsf{sid}) \in F$ set $d_{(\mathcal{F},\mathsf{sid})} = 1$, execute **Round-Update** and return $(\mathsf{clock\text{-}update}, \mathsf{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\mathsf{clock\text{-}read}, \mathsf{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return $(\mathsf{clock\text{-}read}, \mathsf{sid}_C, \tau_{\mathsf{sid}})$ to the requestor (where sid is the session identifier of the calling instance).

Round-Update: For each session sid do: If $d_{(\mathcal{F},\mathsf{sid})} = 1$ for all $\mathcal{F} \in F$ and $d_P = 1$ for all honest parties $P := (\cdot, \mathsf{sid}) \in \mathcal{P}$, then set $\tau_{\mathsf{sid}} = \tau_{\mathsf{sid}} + 1$ and reset $d_{(\mathcal{F},\mathsf{sid})} = 0$ and $d_P = 0$ for all parties $P := (\cdot, \mathsf{sid}) \in \mathcal{P}$.

Figure 2.2: Ideal functionality $\mathcal{G}_{\mathsf{Clock}}$ [KMTZ13, BCH^{+}20].

by [KRDO17]). In the protocols presented in this thesis, we make use of the distributed ledger (i.e., blockchain) functionality $\mathcal{G}_{\mathsf{Ledger}}$ introduced by Badertscher et al. [BMTZ17]. It constitutes the most complete and faithful modeling of a blockchain, and it was shown in [BMTZ17] that the Bitcoin backbone protocol UC-realizes $\mathcal{G}_{\mathsf{Ledger}}$. We show the ideal functionality $\mathcal{G}_{\mathsf{Ledger}}$ below for the sake of completeness and refer to [BMTZ17] for its details. We note that in this thesis we primarily make use of the read and submit interfaces of $\mathcal{G}_{\mathsf{Ledger}}$, i.e., for reading the state and submitting new transactions to the blockchain, respectively.

Ideal Functionality $\mathcal{G}_{\mathsf{Ledger}}$

General: The functionality is parameterized by four algorithms $\mathsf{Validate}, \mathsf{ExtendPolicy}, \mathsf{Blockify}$, and $\mathsf{predict\text{-}time}$, along with two parameters $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$. The functionality manages variables $\mathsf{state}, \mathsf{NxtBC}, \mathsf{buffer}, \tau_L$ and $\vec{\tau}_{\mathsf{state}}$. Initially, $\mathsf{state} := \vec{\tau}_{\mathsf{state}}$, $\mathsf{NxtBC} := \varepsilon$, $\mathsf{buffer} := \emptyset$, $\tau_L = 0$.

Party Management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-)set of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$. The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a new honest party is registered at the ledger, if it is registered with the clock already, then it is added to the party set $\mathcal{H}$ and $\mathcal{P}$, and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to

$\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ and $\mathcal{H}$). The ledger maintains an invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$. A party is considered fully registered if it is registered with the ledger and the clock.

Upon receiving any input I from any party or from the adversary, send $(\mathsf{clock\text{-}read}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and upon receiving response $(\mathsf{clock\text{-}read}, \mathsf{sid}_C, \tau)$ set $\tau_L = \tau$ and do the following:

1. Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of de-synchronized honest parties that have been registered (continuously) since time $\tau' < \tau_L - \mathtt{Delay}$ (to both ledger and clock). Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$. On the other hand, for any synchronize party $P \in \mathcal{H} \setminus \mathcal{P}_{DS}$, if P is not registered to the clock, then $\mathcal{P}_{DS} \cup \{P\}$.

2. If I was received from an honest party $P_i \in \mathcal{P}$:

   - Set $\vec{\mathcal{I}}_H^T = \vec{\mathcal{I}}_H^T \| (I, P, \tau_L)$.
   - Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathsf{NxtBC}, \mathsf{buffer}, \vec{\tau}_{\mathsf{state}})$ and if $\vec{N} \neq \varepsilon$, set $\mathsf{state} := \mathsf{state} \| \mathsf{Blockify}(\vec{N}_1) \| \cdots \| \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathsf{state}} := \vec{\tau}_{\mathsf{state}} \| \tau_L^\ell$, $\tau_L^\ell = \tau_L \| \cdots \| \tau_L$.
   - For each $\mathtt{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 0$, then delete $\mathtt{BTX}$ from buffer. Also, reset $\mathsf{NxtBC} := \varepsilon$.
   - If there exists $P_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathsf{state}| - \mathsf{pt}_j > \mathtt{windowSize}$ or $\mathsf{pt}_j < |\mathsf{state}_j|$, then $\mathsf{pt}_k := |\mathsf{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

3. Depending on the input I and the ID of the sender, execute the respective code:

   - *Submitting a transaction:*
     If $I = (\mathsf{submit}, \mathsf{sid}, \mathtt{tx})$ and is received from a party $P_i \in \mathcal{P}$ or from $\mathcal{S}$ (on behalf of a corrupted party P_i), do the following:
     - Choose a unique transaction ID txid and set $\mathtt{BTX} := (\mathtt{tx}, \mathrm{txid}, \tau_L, P_i)$.
     - If $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \{\mathtt{BTX}\}$.
     - Send $(\mathsf{submit}, \mathtt{BTX})$ to $\mathcal{S}$.
   - *Reading the state:*
     If $I = (\mathsf{read}, \mathsf{sid})$ is received from a fully registered party P, then set $\mathsf{state}_i := \mathsf{state}|_{\min \mathsf{pt}_i, |\mathsf{state}|}$ and return $(\mathsf{read}, \mathsf{sid}, \mathsf{state}_i)$ to the requester. If requester is $\mathcal{S}$, then send $(\mathsf{state}, \mathsf{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{S}$.
   - *Maintaining the ledger state:*
     If $I = (\mathsf{maintain\text{-}ledger}, \mathsf{sid}, \mathrm{minerID})$ is received by an honest party $P_i \in \mathcal{P}$, and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \hat{\tau} > \tau_L$, then send $(\mathsf{clock\text{-}update}, \mathsf{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$. Else, send I to $\mathcal{S}$.
   - *The adversary proposing the next block:*
     If $I = (\mathsf{next\text{-}block}, \mathrm{hFlag}, (\mathrm{txid}_1, \ldots, \mathrm{txid}_\ell))$ is sent from the adversary, update NxtBC as follows:
     - Set $\mathrm{listOfTxid} \leftarrow \epsilon$.
     - For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} := (x, \mathrm{txid}, \mathrm{minerID}, \tau_L, P_i) \in \mathsf{buffer}$ with ID $\mathrm{txid} = \mathrm{txid}_i$, then set $\mathrm{listOfTxid} := \mathrm{listOfTxid} \| \mathrm{txid}_i$.

  - Finally, set $\mathsf{NxtBC} := \mathsf{NxtBC} \| (\text{hFlag}, \text{listOfTxid})$ and output $(\mathsf{next\text{-}block}, ok)$ to $\mathcal{S}$.

- *The adversary setting state-slackness:*
  If $I = (\mathsf{set\text{-}slack}, (P_{i_1}, \widehat{\mathsf{pt}}'_{i_1}), \ldots, (P_{i_\ell}, \widehat{\mathsf{pt}}'_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{S}$, do the following:
  - If for all $j \in [\ell]$: $|\mathsf{state}| - \widehat{\mathsf{pt}}_{i_j} \leq \texttt{windowSize}$ and $\mathsf{pt}_{i_1} := \widehat{\mathsf{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\mathsf{set\text{-}slack}, ok)$ to $\mathcal{S}$.
  - Otherwise, set $\mathsf{pt}_{i_j} := |\mathsf{state}|$ for all $j \in [\ell]$.

- *The adversary setting the state for de-synchronized parties:*
  If $I = (\mathsf{desync\text{-}state}, (P_{i_1}, \mathsf{state}'_{i_1}), \ldots, (P_{i_\ell}, \mathsf{state}'_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ from the adversary $\mathcal{S}$, set $\mathsf{state}_{i_j} = \mathsf{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\mathsf{desync\text{-}state}, ok)$ to $\mathcal{S}$.

Cryptocurrencies can be built on top of the aforedescribed distributed ledger (i.e., blockchain) infrastructure by enforcing that the transactions submitted have a special format. More precisely, we have that the transactions encode information such as a sending address aid_S, a recipient address aid_R and a value v of the amount of coins transferred. At a high level, an address has an associated predicate that needs to be satisfied in order for the coins to be spent from this address. Then, the parties that participate in the blockchain network validate the transactions by checking if the associated predicates are satisfied, and only the valid transactions are added to the blockchain at the end. For example, the sending address aid_S may have an associated predicate that is a signature verification algorithm $\Sigma.\mathsf{Ver}$ with respect to a public key pk_S. The sender would need to produce a valid signature σ_S on the transaction $\texttt{tx}$ that is verifiable with respect to pk_S, i.e., $\Sigma.\mathsf{Ver}(\mathsf{pk}_S, \texttt{tx}, \sigma_S) = 1$. The transaction $\texttt{tx}$ would also include the predicate associated with the recipient address aid_R that needs to be satisfied in order to spend the coins from aid_R. In the cryptocurrency literature these predicates are referred to as *scripts*. These scripts can range anywhere from complex spending conditions captured by Turing complete programs, referred as *smart contracts*, to simple hash-based contracts, such as *Hash Time-Lock Contracts (HTLCs)*, which postulate that the coins from an address can be spent if the spender provides a pre-image of some specific hash value. In the rest of this thesis we only consider scripts that can be encoded with a plain and adaptor signature schemes.

Payment Channels. A *payment channel* is an off-chain payment protocol that allows the involved parties to make several high frequency payments, while only registering the initial and final balances on the chain. In order to do this, a payment channel protocol proceeds in three phases.

In the first phase, referred as *channel opening*, the parties Alice and Bob open a payment channel γ_{AB} by posting a single transaction on chain. This results in a shared address aid_{AB} between Alice and Bob, where some funds are deposited for some amount of time t. Hence, both Alice and Bob have to agree on a transaction to spend from the channel γ_{AB} before the time t. After the time t, the funds from aid_{AB} are reimbursed to Alice and Bob.

The second phase, referred as *channel update*, involves Alice and Bob sending each other coins (in the form of transactions) from the joint address aid_{AB}, which updates the corresponding balances of the parties within the channel γ. We note that these constitute valid payment transactions, but they are not posted on the chain, and instead are stored locally by updating the channel state.

The third phase, referred as *channel closing*, finalizes the payments between Alice and Bob. This is done by one of the parties posting the most recent channel state on the blockchain. This effectively distributes the funds between Alice and Bob according to the latest channel state, and the channel is considered closed. As we can observe, no matter the number payments between Alice and Bob, only two transactions go on chain. This boosts the scalability of payments and leads to increased transaction processing rates. Moreover, payment channels have been implemented and used in practice, where Lightning Network built on top of Bitcoin and Raiden Network built on top of Ethereum are some prominent examples.

We defer the formal description and construction of payment channels to Section 6.1.

CHAPTER 3

Post-Quantum Adaptor Signatures

In this chapter we present a post-quantum adaptor signature construction from isogenies, which is based on CSI-FiSh signature scheme [BKV19]. First, in Section 3.1 we recall CSI-FiSh signature scheme, and then in Section 3.2 we provide our post-quantum adaptor signature construction. In Section 3.3 we formally prove the security of our construction in (quantum) random oracle model, and lastly in Section 3.4 we evaluate the performance of our construction.

3.1 CSI-FiSh Signature Scheme

CSI-FiSh is a signature scheme [BKV19] obtained by applying Fiat-Shamir transform to an identification scheme. First, we recall the interactive zero-knowledge identification scheme, where a prover wants to convince a verifier that it knows a secret element $\mathfrak{a} \in \mathrm{Cl}(\mathcal{O})$ of its public key $E_a = \mathfrak{a} \star E_0$, for $\mathfrak{a} = \mathfrak{g}^a$ and $a \in \mathbb{Z}_N$, where E_0 is a publicly known base curve. The scheme is as follows:

- Prover samples a random $\mathfrak{b} = \mathfrak{g}^b$ for $b \in \mathbb{Z}_N$ and commits to $E_b = [b]E_0$ (this corresponds to $E_b = \mathfrak{b} \star E_0$ with our notation).
- Verifier samples a random challenge bit $c \in \{0, 1\}$.
- If $c = 0$, prover replies with $r = b$, otherwise it replies with $r = b - a \bmod N$ (reducing modulo N to avoid any leakage on a).
- If $c = 0$, verifier verifies that $E_b = [r]E_0$, otherwise verifies that $E_b = [r]E_a$.

This scheme is clearly correct, and it has soundness 1/2. For the zero-knowledge property, it is important that elements in $\mathrm{Cl}(\mathcal{O})$ can be sampled uniformly, and that they have unique representation.

In order to improve soundness, the authors of [BKV19] increased the size of the public key. For a positive integer S, the secret key becomes the vector $(a_1, \ldots, a_{S-1})$ of dimension

$S-1$, and public key is set to $(E_0, E_1 = [a_1]E_0, \dots, E_{S-1} = [a_{S-1}]E_0)$. Then, the prover proves to the verifier that it knows an $s \in \mathbb{Z}_N$, such that $[s]E_i = E_j$ for some pair of curves in the public key (with $i \neq j$). In order to further increase the challenge space, one can exploit the fact that given a curve $E = [a]E_0$, its quadratic twist E^t, which can be computed very efficiently, is $\mathcal{F}_p$-isomorphic to $[-a]E_0$. Therefore, one can almost double the set of public key curves going from $E_0, E_1, \dots, E_{S-1}$ to $E_{-S+1}, \dots, E_0, \dots, E_{S-1}$, where $E_{-i} = E_i^t$, without any increase in communication cost. Combining all these the soundness error drops to $\frac{1}{2S-1}$. To achieve security level λ (i.e., $2^{-\lambda}$ soundness error), we need to repeat the protocol $t_S = \lambda / \log_2(2S-1)$ times.

The described identification scheme when combined with the Fiat-Shamir heuristic, for a hash function $\mathcal{H}\colon \{0,1\}^* \to \{-S+1, \dots, S-1\}^{t_S}$, gives the CSI-FiSh signature scheme shown in Algorithm 1, where sign denotes the sign of the integer. In [BKV19] it is shown that CSI-FiSh is SUF-CMA secure under the MT-GAIP assumption, when $\mathcal{H}$ is modeled as a quantum random oracle, hence, it is strongly unforgeable in the quantum random oracle model (QROM) [DFMS19].

Algorithm 1 CSI-FiSh Signature

```
Public parameters: base curve $E_0$, class number $N = \#\mathrm{Cl}(\mathcal{O})$, security parameters
  $\lambda, t_S, S$, hash function $\mathcal{H}\colon \{0,1\}^* \to \{-S+1, \dots, S-1\}^{t_S}$
procedure KeyGen($1^\lambda$)
    for $i \in \{1, \dots, S-1\}$ do
        $a_i \leftarrow\$\ \mathbb{Z}_N$
        $E_i \leftarrow [a_i]E_0$
    Set sk := $[a_i\colon i \in \{1, \dots, S-1\}]$
    Set pk := $[E_i\colon i \in \{1, \dots, S-1\}]$
    return (sk, pk)
procedure Sig(sk, $m$)
    Parse sk as $(a_1, \dots, a_{S-1})$
    $a_0 \leftarrow 0$
    for $i \in \{1, \dots, t_S\}$ do
        $b_i \leftarrow \mathbb{Z}_N$
        $E'_i \leftarrow [b_i]E_0$
    $(c_1, \dots, c_{t_S}) = \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$
    for $i \in \{1, \dots, t_S\}$ do
        $r_i \leftarrow b_i - \mathsf{sign}(c_i) a_{|c_i|} \bmod N$
    return $\sigma := (r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
procedure Ver(pk, $m$, $\sigma$)
    Parse pk as $(E_1, \dots, E_{S-1})$
    Parse $\sigma$ as $(r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
    Define $E_{-i} := E_i^t$ for all $i \in [1, S-1]$
    for $i \in \{1, \dots, t_S\}$ do
        $E'_i \leftarrow [r_i]E_{c_i}$
    $(c'_1, \dots, c'_t) = \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$
    if $(c_1, \dots, c_{t_S}) == (c'_1, \dots, c'_{t_S})$ then
        return 1
    else
        return 0
```

3.1.1 Zero-Knowledge Proof for Group Actions

Cozzo and Smart [CS20] showed how to prove knowledge of a secret isogeny generically. In detail, they showed a zero-knowledge proof for the following relation:

$$L_j := \left\{ \Big((E_1, E'_1, \dots, E_j, E'_j), s \Big) \colon \bigwedge_{i=1}^{j} (E'_i = [s]E_i) \right\}.$$

Intuitively, the prover wants to prove in zero-knowledge that it knows a unique witness s for j simultaneous instances of the GAIP. In [CS20] two variants of such a proof are given, one when $E_1 = \cdots = E_j = E_0$, called Special case with soundness error $1/3$, and another one when that condition does not hold, called General case with soundness error $1/2$. In our paper we only need the General case for $j = 2$. Since the proof has soundness error of $1/2$, we need to repeat it $t_{ZK} = \lambda$ times to achieve a security level of λ. Using a "slow" hash function $\mathcal{F}$, as in CSI-FiSh, which is 2^k times slower than a normal hash function we can reduce the number of repetitions to $t_{ZK} = \lambda - k$. For example, when setting $\lambda = 128$ and $k = 16$, as in the fastest CSI-FiSh parameters, we get $t_{ZK} = 112$. In the random oracle model the proof can be made non-interactive using a hash function $\mathcal{F}$ with codomain $\{0,1\}^{t_{ZK}}$. For brevity, we only present the non-interactive single iteration (i.e., $t_{ZK} = 1$) variant of the proof for L_j in Algorithm 2.

Algorithm 2 Non-interactive zero-knowledge proof for L_j

```
Public parameters: class number $N = \#\mathrm{Cl}(\mathcal{O})$, hash function $\mathcal{F}\colon \{0,1\}^* \to \{0,1\}$
procedure $\Pi_{\mathsf{NIZK}}.\mathsf{P}(x, s)$
    Parse $x$ as $(E_1, E_1', \ldots, E_j, E_j')$
    $b \leftarrow\$ \mathbb{Z}_N$
    for $i \in \{1, \ldots, j\}$ do
        $\hat{E}_i \leftarrow [b]E_i$
    $c = \mathcal{F}(E_1 \| E_1' \| \hat{E}_1 \| \cdots \| E_j \| E_j' \| \hat{E}_j)$
    $r \leftarrow b - c \cdot s \bmod N$
    return $\pi := ((\hat{E}_1, \ldots, \hat{E}_j), r)$
procedure $\Pi_{\mathsf{NIZK}}.\mathsf{V}(x, \pi)$
    Parse $x$ as $(E_1, E_1', \ldots, E_j, E_j')$
    Parse $\pi$ as $((\hat{E}_1, \ldots, \hat{E}_j), r)$
    $c = \mathcal{F}(E_1 \| E_1' \| \hat{E}_1 \| \cdots \| E_j \| E_j' \| \hat{E}_j)$
    if $c = 0$ then
        return $\bigwedge_{i=1}^{j}([r]E_i = \hat{E}_i)$
    else if $c = 1$ then
        return $\bigwedge_{i=1}^{j}([r]E_i' = \hat{E}_i)$
```

3.2 IAS Adaptor Signature Scheme

Despite the fact that CSI-FiSh is simply a signature scheme obtained by applying Fiat-Shamir to multiple repetitions of Schnorr-type identification scheme from isogenies, one cannot trivially construct a Schnorr-type AS as described in [AEE+21].

Strawman Approach. Let us consider a single iteration of the identification scheme (i.e., $t_S = 1$), and a hard relation $R^1_{E_0} \subseteq \mathcal{E} \times \mathrm{Cl}(\mathcal{O})$, for a set of elliptic curves $\mathcal{E}$, to be defined as $R^1_{E_0} := \{(E_Y, y) \mid E_Y = [y]E_0\}$. A naïve approach to construct an AS from a single-iteration CSI-FiSh, following the Schnorr AS from [AEE+21], is to compute the randomness inside the pre-signature algorithm as $E' \leftarrow [b]E_Y$ instead of doing $E' \leftarrow [b]E_0$ as in the original construction, and leave the rest of the algorithm identical to the signing algorithm of CSI-FiSh. However, later during the pre-verification, given the pre-signature $\hat{\sigma} := (\hat{r}, c)$, the statement E_Y and c-th public key E_c, one cannot verify the correctness of the pre-signature $\hat{\sigma}$. More concretely, we have that $\hat{r} = b - \mathsf{sign}(c)a_{|c|} \bmod N$, $E_c = [\mathsf{sign}(c)a_{|c|}]E_0$ and $E_Y = [y]E_0$. Now, using these values we can compute $\hat{E}' = [\hat{r}]E_c = [b]E_0$, but then we cannot combine $\hat{E}'$ with E_Y to obtain $E' = [b]E_Y$, which we need for verification. Analogous problem happens if we first compute the group action $\hat{E}' = [r]E_Y$, and then try to combine it with E_c to obtain the desired E'. The reason behind this problem is that we have a limited algebraic structure. More precisely, the group action is defined as $\star\colon \mathrm{Cl}(\mathcal{O}) \times \mathcal{E} \to \mathcal{E}$, for class group $\mathrm{Cl}(\mathcal{O})$ and set of elliptic curves $\mathcal{E}$. This implies that we can pair a class group element with an elliptic curve to map it to a new elliptic curve, however, we do not have any meaningful operation over the set $\mathcal{E}$ that would allow us to purely pair two elliptic curves and map to a third one.

3.2.1 Our Construction

On a high-level, we have to circumvent the limited algebraic structure of CSI-FiSh, which prevents us from extracting the randomness. We solve this problem by means of a zero-knowledge proof showing the validity of the pre-signature construction. This might remind of the ECDSA- based AS construction by Aumayr et al. [AEE$^+$21], where a zero-knowledge proof is also used to prove the consistency of the randomness, which would not be otherwise possible due to the lack of linearity of ECDSA. Besides not being post-quantum secure, their cryptographic construction (i.e., the underlying signature scheme and thus the resulting zero-knowledge proof) is, however, fundamentally different because the issue in CSI-FiSh is a limited algebraic structure as opposed to a lack of linearity as in ECDSA.

More concretely, to compute the pre-signature for E_Y, the signer samples a random $b \leftarrow\!\$\ \mathbb{Z}_N$, computes $\hat{E}' \leftarrow [b]E_0$ and $E' \leftarrow [b]E_Y$. Then, the signer uses E' as input to the hash function to compute the challenge c, and also includes E' as part of the pre-signature. Lastly, to ensure that the same value b is used in computation of both $\hat{E}'$ and E', a zero-knowledge proof π that $(E_0, \hat{E}', E_Y, E') \in L_2$ is attached to the pre-signature (see Section 3.1.1 for such a proof). So, the pre-signature looks like $\hat{\sigma} := (\hat{r}, c, \pi, E')$. The pre-signature verification of $\hat{\sigma}$ then involves extracting $\hat{E}'$ by computing the group actions $[\hat{r}]E_c$, using it to verify the proof π, and finally, checking that the hash of E' produces the expected challenge c. The pre-signature adaptation is done by adding the corresponding witness y to $\hat{r}$ of the pre-signature to obtain the full valid signature $\sigma := (r, c)$. In an opposite manner, the extraction is done by subtracting r of the valid signature from $\hat{r}$ of the pre-signature.

Since CSI-FiSh involves multiple iterations (more concretely t_S iterations), we extend the hard relation $R^1_{E_0}$ to $R^{t_S}_{E_0} \subseteq \mathcal{E}^{t_S} \times \mathrm{Cl}(\mathcal{O})^{t_S}$, to be defined as $R^{t_S}_{E_0} := \{(\vec{E}_Y := (E^1_Y, \ldots, E^{t_S}_Y), \vec{y} := (y_1, \ldots, y_{t_S})) \mid E^i_Y = [y_i]E_0 \text{ for all } i \in [1, t_S]\}$, and apply the above described method to every iteration with a different E^i_Y.

Although, the described scheme achieves correctness, one cannot prove its security directly. As we would like to reduce both the unforgeability and witness extractability of the scheme to the strong unforgeability of CSI-FiSh, inside the reduction we need a way to answer the pre-signature queries by only relying on the signing oracle of CSI-FiSh, and without access to the secret key sk or the witness $(y_1, \ldots, y_{t_S})$. In order to overcome this issue, we use a modified hard relation. Let $R^*_{E_0}$ consist of pairs $I_Y := (\vec{E}_Y, \pi_Y)$, where $\vec{E}_Y \in L_{R^{t_S}_{E_0}}$

Algorithm 3 Adaptor Signature $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}(\mathsf{IAS})$

```
Public parameters: base curve $E_0$, class number $N = \#\mathrm{Cl}(\mathcal{O})$, security parameters
$\lambda, t_S, S$, hash function $\mathcal{H}\colon \{0,1\}^* \to \{-S+1, \dots, S-1\}^{t_S}$
procedure $\mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
    Parse $\mathsf{sk}$ as $(a_1, \dots, a_{S-1})$
    Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
    Parse $\vec{E}_Y$ as $(E_Y^1, \dots, E_Y^{t_S})$
    $a_0 \leftarrow 0$
    for $i \in \{1, \dots, t_S\}$ do
        $b_i \leftarrow \mathbb{Z}_N$
        $\hat{E}'_i \leftarrow [b_i]E_0$
        $E'_i \leftarrow [b_i]E_Y^i$
        Set $x_i := (E_0, \hat{E}'_i, E_Y^i, E'_i)$
        $\pi_i \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}(x_i, b_i)$
    $(c_1, \dots, c_{t_S}) = \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$
    for $i \in \{1, \dots, t_S\}$ do
        $\hat{r}_i \leftarrow b_i - \mathsf{sign}(c_i) a_{|c_i|} \bmod N$
    return $\hat{\sigma} := (\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots,$
            $c_{t_S}, \pi_1, \dots, \pi_{t_S}, E'_1, \dots, E'_{t_S})$
procedure $\mathsf{PreVer}(\mathsf{pk}, m, I_Y, \hat{\sigma})$
    Parse $\mathsf{pk}$ as $(E_1, \dots, E_{S-1})$
    Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
    Parse $\vec{E}_Y$ as $(E_Y^1, \dots, E_Y^{t_S})$
    Parse $\hat{\sigma}$ as $(\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S},$
            $\pi_1, \dots, \pi_{t_S}, E'_1, \dots, E'_{t_S})$
    Set $E_{-i} = E_i^t$ for all $i \in [1, S-1]$
    for $i \in \{1, \dots, t_S\}$ do
        $\hat{E}'_i \leftarrow [\hat{r}_i]E_{c_i}$
        Set $x_i := (E_0, \hat{E}'_i, E_Y^i, E'_i)$
        if $\Pi_{\mathsf{NIZK}}.\mathsf{V}(x_i, \pi_i) \neq 1$ then
            return 0
    if $(c_1, \dots, c_{t_S}) == \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$
then
        return 1
    else
        return 0
procedure $\mathsf{Ext}(\sigma, \hat{\sigma}, I_Y)$
    Parse $\sigma$ as $(r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
    Parse $\hat{\sigma}$ as $(\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S},$
            $\pi_1, \dots, \pi_{t_S}, E'_1, \dots, E'_{t_S})$
    for $i \in \{1, \dots, t_S\}$ do
        $y'_i \leftarrow r_i - \hat{r}_i$
    Set $\vec{y}' := [y'_i \colon i \in \{1, \dots, t_S\}]$
    if $(I_Y, \vec{y}') \notin R^*_{E_0}$ then
        return $\perp$
    return $\vec{y}'$
procedure $\mathsf{Adapt}(\hat{\sigma}, \vec{y})$
    Parse $\hat{\sigma}$ as $(\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S},$
            $\pi_1, \dots, \pi_{t_S}, E'_1, \dots, E'_{t_S})$
    Parse $\vec{y}$ as $(y_1, \dots, y_{t_S})$
    for $i \in \{1, \dots, t_S\}$ do
        $r_i \leftarrow \hat{r}_i + y_i \bmod N$
    return $\sigma := (r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
```

is as previously defined, and π_Y is a non-interactive zero-knowledge proof that $\vec{E}_Y \in L_{R^{t_S}_{E_0}}$. Formally, we have that $R^*_{E_0} := \{((\vec{E}_Y, \pi_Y), \vec{y}) \mid \vec{E}_Y \in L_{R^{t_S}_{E_0}} \wedge \Pi_{\mathsf{NIZK}}.\mathsf{V}(\vec{E}_Y, \pi_Y) = 1\}$. Due to the soundness of the proof system, if $R^{t_S}_{E_0}$ is a hard relation, then so is $R^*_{E_0}$. Since we are in the random oracle model, the reduction then can use the random oracle query table to extract a witness from the proof π_Y, and answer the pre-signature oracle queries using this witness.

The resulting AS scheme, which we denote as $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$ and call as IAS, is depicted in Algorithm 3. The security of our construction is captured by the following theorem, which we formally prove in Section 3.3.

Theorem 1. *Let Π_{NIZK} be a NIZKPoK with an online extractor, the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ be* SUF-CMA *secure and $R^*_{E_0}$ be a hard relation, then the adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$, as defined in Algorithm 3, is secure in QROM.*

Optimization. Our construction, as defined in Algorithm 3, makes sure that all t_S parts of the signature are adapted (i.e., each r_i, for $i \in \{1, \ldots, t_S\}$, is adapted). This is due to the fact that IAS is based on CSI-FiSh, which in turn is constructed from multiple iterations of a Schnorr-type identification scheme as described in Section 3.1. However, this also points to the fact that CSI-FiSh is just a much less efficient version of Schnorr. Therefore, one can have a more efficient variant of IAS by only adapting one of the iterations (e.g., the first iteration). In this variant, during the pre-signature algorithm we compute π_1 and E'_1 using E^1_Y as defined in Algorithm 3, and attach them to the pre-signature $\hat{\sigma}$ as before. But, for the rest of the iterations (i.e., for $i \in \{2, \ldots, t_S\}$), we do not compute any zero-knowledge proof, and compute E'_i using E_0 as done in the signing algorithm of CSI-FiSh (see Algorithm 1). This means that the pre-signature $\hat{\sigma}$ is only incomplete in the first component (i.e., only $\hat{r}_1$ needs to be adapted to obtain a valid signature). Hence, the extraction and adaptation only depend on the first component of the pre-signature/signature pair. Using this approach we revert from the hard relation $R^{t_S}_{E_0}$ to $R^1_{E_0}$, and define a new modified relation $R^{\dagger}_{E_0}$, which consists of pairs $I_Y := (E_Y, \pi_Y)$, such that $E_Y \in L_{R^1_{E_0}}$ and π_Y is a zero-knowledge proof that $E_Y \in L_{R^1_{E_0}}$. More formally, we have that $R^{\dagger}_{E_0} := \{((E_Y, \pi_Y), y) \mid E_Y \in L_{R^1_{E_0}} \wedge \Pi_{\mathsf{NIZK}}.\mathsf{V}(E_Y, \pi_Y) = 1\}$. Due to the soundness of the proof system, if $R^1_{E_0}$ is a hard relation, then so is $R^{\dagger}_{E_0}$. We call this optimized variant $\mathsf{O\text{-}IAS}$, and capture its security with the following theorem, which we formally proof in Section 3.3.

Theorem 2. *Let Π_{NIZK} be a NIZKPoK with an online extractor, the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ be* SUF-CMA *secure and $R^{\dagger}_{E_0}$ br a hard relation, the adaptor signature scheme $\Xi_{R^{\dagger}_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$, is secure in QROM.*

Remark 1. Although in this work we specifically focused on CSI-FiSh signature scheme, we note that our techniques to construct an adpaptor signature scheme can also be applied to other isogeny-based signatures that have similar algebraic limitations, such as the recently proposed SQISign [DKL+20, DLRW23] signature scheme.

Remark 2. We also remark that our constructions cannot be turned to a two-party adaptor signature with aggregatable keys (as defined in Definition 17) using the transformation of Erwig et al. [EFH+21] while there exists no known public key aggregation technique in isogeny-based cryptography.

3.3 Security Proof

In this section we formally prove the security of IAS. We recall the theorem stated in Section 3.2, which we prove here.

Theorem 1. *Let Π_{NIZK} be a NIZKPoK with an online extractor, the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ be $\mathsf{SUF\text{-}CMA}$ secure and $R^*_{E_0}$ be a hard relation, then the adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$, as defined in Algorithm 3, is secure in QROM.*

Proof. We begin by proving that the adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$ (IAS) satisfies pre-signature adaptability. In fact, we prove a slightly stronger statement, which says that any valid pre-signature adapts to a valid signature with probability 1.

Lemma 1 (Pre-signature Adaptability)**.** *The adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$ satisfies pre-signature adaptability.*

Proof. Let us fix some arbitrary $(I_Y, \vec{y}) \in R^*_{E_0}$, $m \in \{0,1\}^*$, $\mathsf{pk} := (E_1, \ldots, E_{S-1}) \in \mathcal{E}^{S-1}$ and $\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S}, \pi_1, \ldots, \pi_{t_S}, E'_1, \ldots, E'_{t_S}) \in \mathbb{Z}_N^{t_S} \times \{-S+1, \ldots, S-1\}^{t_S} \times (\{0,1\}^*)^{t_S} \times \mathcal{E}^{t_S}$. Let $(c_1, \ldots, c_{t_S}) = \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$ and for all $i \in \{1, \ldots, t_S\}$,

$$\hat{E}'_i \leftarrow [\hat{r}_i] E_{c_i}.$$

Assuming that $\mathsf{PreVer}(\mathsf{pk}, m, I_Y, \hat{\sigma}) = 1$, we know that there exists $(b_1, \ldots, b_{t_S}) \in \mathbb{Z}_N^{t_S}$ s.t. for all $i \in \{1, \ldots, t_S\}$, $\hat{E}'_i := [b_i]E_0$ and $E'_i := [b_i]E^i_Y$ for $I_Y := (\vec{E}_Y := (E^1_Y, \ldots, E^{t_S}_Y), \pi_Y)$. Moreover, by the definition of Adapt, we know that $\mathsf{Adapt}(\hat{\sigma}, \vec{y} := (y_1, \ldots, y_{t_S})) = (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$ for $r_i := \hat{r}_i + y_i$ for all $i \in \{1, \ldots, t_S\}$. Hence, we have

$$\begin{aligned}
\mathcal{H}([r_1]E_{c_1} \| \cdots \| [r_{t_S}]E_{c_{t_S}} \| m) &= \mathcal{H}([y_1][\hat{r}_1]E_{c_1} \| \cdots \| [y_{t_S}][\hat{r}_{t_S}]E_{c_{t_S}} \| m) \\
&= \mathcal{H}([y_1]\hat{E}'_1 \| \cdots \| [y_{t_S}]\hat{E}'_{t_S} \| m) \\
&= \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m) \\
&= (c_1, \ldots, c_{t_S}).
\end{aligned}$$

□

Lemma 2 (Pre-signature Correctness)**.** *The adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$ satisfies pre-signature correctness.*

Proof. Let us fix some arbitrary $(\mathsf{sk} := (a_1, \ldots, a_{t_S}), \vec{y}) \in \mathbb{Z}_N^{2 \cdot t_S}$ and $m \in \{0,1\}^*$, compute $E_i \leftarrow [a_i]E_0$ and $E^i_Y \leftarrow [y_i]E_0$ for all $i \in \{1, \ldots, t_S\}$, set $\mathsf{pk} := (E_1, \ldots, E_{t_S})$, compute $\pi_Y \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}((E_0, E^1_Y, \ldots, E_0, E^{t_S}_Y), y)$ and set $I_Y := (\vec{E}_Y, \pi_Y)$. For $\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S}, \pi_1, \ldots,\ \pi_{t_S}, E'_1, \ldots, E'_{t_S}) \leftarrow \mathsf{PreSig}(\mathsf{sk},\ m, I_Y)$, and for all $i \in \{1, \ldots, t_S\}$, it holds that $\hat{E}'_i = [b_i]E_0$, $E'_i = [b_i]E^i_Y$, $(c_1, \ldots, c_{t_S}) = \mathcal{H}(E'_1 \| \cdots \| E'_{t_S} \| m)$ and $\hat{r}_i = b_i - \mathsf{sign}(c_i) a_{|c_i|} \bmod N$. Set for all $i \in \{1, \ldots, t_S\}$,

$$\hat{E}'_i := [\hat{r}_i]E_{c_i} = [b_i]E_0.$$

By correctness of Π_{NIZK} we know that $\Pi_{\mathsf{NIZK}}.\mathsf{V}((E_0, \hat{E}_i, E^i_Y, E'_i), \pi_i) = 1$, and hence, we have that $\mathsf{PreVer}(\mathsf{pk}, m, I_Y, \hat{\sigma}) = 1$. By Lemma 1, this implies that $\mathsf{Ver}(\mathsf{pk}, m, \sigma) = 1$ for

$\sigma = (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S}) := \mathsf{Adapt}(\hat{\sigma}, \vec{y} := (y_1, \ldots, y_{t_S}))$. From definition of Adapt, we know that $r_i = \hat{r}_i + y_i$ for all $i \in \{1, \ldots, t_S\}$, and hence,

$$\mathsf{Ext}(\sigma, \hat{\sigma}, I_Y) = r_i - \hat{r}_i = (\hat{r}_i + y_i) - \hat{r}_i = y_i \text{ for all } i \in \{1, \ldots, t_S\}.$$

□

Lemma 3 (aEUF-CMA Security)**.** *Assuming that the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ is* SUF-CMA *secure and $R^*_{E_0}$ is a hard relation, the adaptor signature scheme $\Xi_{R^*_{E_0}, \Sigma_{\mathsf{CSI\text{-}FiSh}}}$, as defined in Algorithm 3, is* aEUF-CMA *secure.*

Proof. We prove the adaptor unforgeability by reduction to strong unforgeability of the CSI-FiSh signatures scheme, which was proved in [BKV19] to hold in quantum random oracle model (QROM) [DFMS19]. We consider an adversary $\mathcal{A}$ who plays the aSigForge game, and then we build a simulator $\mathcal{S}$ (i.e., a reduction) that plays the strong unforgeability experiment for the CSI-FiSh signature scheme and uses $\mathcal{A}$'s forgery in aSigForge to win its own experiment. $\mathcal{S}$ has access to the signing oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and the random oracle $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$, which it uses to simulate oracle queries for $\mathcal{A}$, namely random oracle ($\mathcal{H}$), 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 the full signatures into pre-signatures for $\mathcal{A}$. In order to do so, the reduction faces two challenges: 1) $\mathcal{S}$ needs to learn the witness $\vec{y}$ for the statement $\vec{E}_Y$ for which the pre-signature is supposed to be generated, and 2) $\mathcal{S}$ needs to simulate the zero-knowledge proofs π_i, for $\{1, \ldots, t_S\}$, which proves the consistency of the randomnesses in the pre-signature.

More precisely, upon receiving a $\mathcal{O}_{\mathrm{pS}}$ query from $\mathcal{A}$ on input a message m and an instance $I_Y := (\vec{E}_Y, \pi_Y)$, $\mathcal{S}$ queries its signing oracle to obtain a full signature on m. Then, $\mathcal{S}$ needs to learn a witness $\vec{y}$, s.t. $E^i_Y = [y_i]E_0$ for $i \in \{1, \ldots, t_S\}$, 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 $\vec{y}$, and consequently transform a full signature into a valid pre-signature. Additionally, since a valid pre-signature contains t_s zero-knowledge proofs for L_2 (see Section 3.1.1), the simulator has to simulate these proof without knowledge of the corresponding witness. In order to achieve this, we make use of the zero-knowledge property, which allows for simulation of a proof for a statement without knowing the corresponding witness.

Game $\boldsymbol{G_0}$: This game corresponds to the original aSigForge game, where the adversary $\mathcal{A}$ has to come up with a valid forgery for a message m of its 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 $\mathcal{H}$. It trivially follows that

$$\Pr[\boldsymbol{G_0} = 1] = \Pr[\mathsf{aWitExt}_{\mathcal{A}, \Xi_{R^*_{E_0}, \Sigma_{\mathsf{CSI\text{-}FiSh}}}}(\lambda) = 1].$$

```
$\boldsymbol{G_0}$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$m \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
$(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
$\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
$b := \mathsf{Ver}(\mathsf{pk}, m, \sigma^*)$
$\textbf{return } (m \notin \mathcal{Q} \ \wedge \ b)$

$\mathcal{O}_{\mathrm{S}}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$\textbf{return } \sigma$

$\mathcal{H}(x)$
$\textbf{if } H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
$\textbf{return } H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$\textbf{return } \hat{\sigma}$
```

Figure 3.1: The formal definition of game $\boldsymbol{G_0}$.

Game $\boldsymbol{G_1}$: This game 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 $\hat{\sigma}$ using the witness $\vec{y}$ results in σ^*. In that case, the game aborts.

Claim 1. *Let* Bad_1 *be the event that* $\boldsymbol{G_1}$ *aborts, then it holds that* $\Pr[\mathsf{Bad}_1] \leq \mathsf{negl}(\lambda)$.

Proof. We prove this claim using a reduction to the hardness of the relation $R^*_{E_0}$. More precisely, we construct a simulator $\mathcal{S}$ that breaks the hardness of the relation assuming it has access to an adversary $\mathcal{A}$ that causes $\boldsymbol{G_1}$ to abort with non-negligible probability. $\mathcal{S}$ gets a challenge I^*_Y, upon which it generates a key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ in order to simulate $\mathcal{A}$'s queries to the oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$. The 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 $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I^*_Y)$ and returns the pair $(\hat{\sigma}, I^*_Y)$ to the adversary which outputs a forgery σ. Assuming that Bad_1 happened (i.e., $\mathsf{Adapt}(\hat{\sigma}, \vec{y}) = \sigma$), we know that due to the correctness property, the simulator can extract $\vec{y}^*$ by executing $\mathsf{Ext}(\sigma, \hat{\sigma}, I^*_Y)$ to obtain a valid statement/witness pair for the relation $R^*_{E_0}$ (i.e., $(I^*_Y, \vec{y}^*) \in R^*_{E_0}$).

We note that the view of $\mathcal{A}$ is indistinguishable to his view in $\boldsymbol{G_1}$, since the challenge I^*_Y is an instance of the hard relation $R^*_{E_0}$, and therefore, 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 event Bad_1 happening. By our assumption, this is non-negligible, which is a contradiction to the hardness of $R^*_{E_0}$. □

```
$\boldsymbol{G_1}$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$m^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
$(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m^*, I_Y)$
$\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
if $\mathsf{Adapt}(\hat{\sigma}, \vec{y}) = \sigma^*$
    abort
$b := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
return $(m^* \notin \mathcal{Q} \ \wedge \ b)$

$\mathcal{O}_{\mathrm{S}}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$

$\mathcal{H}(x)$
if $H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\$\ \mathcal{C}^{t_S}$
return $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$
$\hat{\sigma} \leftarrow \mathsf{Sig}(\mathsf{sk}, m, I_Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\hat{\sigma}$
```

Figure 3.2: The formal definition of game $\boldsymbol{G_1}$.

Since games $\boldsymbol{G_1}$ and $\boldsymbol{G_0}$ are equivalent except when event Bad_1 happens, it holds that

$$\Pr[\boldsymbol{G_1} = 1] \leq \Pr[\boldsymbol{G_0} = 1] + \mathsf{negl}(\lambda).$$

Game $\boldsymbol{G_2}$: The only changes between $\boldsymbol{G_1}$ and $\boldsymbol{G_2}$ are for the $\mathcal{O}_{\mathrm{pS}}$ oracle. More precisely, during the $\mathcal{O}_{\mathrm{pS}}$ queries, this game extracts a witness $\vec{y}$ by executing the extractor algorithm $\mathcal{K}$ on input the statement $\vec{E}_Y$, the proof π_Y and the list of random oracle queries H. If for the extracted witness $\vec{y}$ it does not hold that $((\vec{E}_Y, \pi_Y), \vec{y}) \in R^*_{E_0}$, then the game aborts.

Claim 2. *Let* Bad_2 *be the event that* $\boldsymbol{G_2}$ *aborts during an* $\mathcal{O}_{\mathrm{pS}}$ *execution, then it holds that* $\Pr[\mathsf{Bad}_2] \leq \mathsf{negl}(\lambda)$.

Proof. According to the *online extractor* property of Π_{NIZK}, for a witness $\vec{y}$ extracted from a proof π_Y of statement $\vec{E}_Y$ such that $\Pi_{\mathsf{NIZK}}.\mathsf{V}(\vec{E}_Y, \pi_Y) = 1$, it holds that $((\vec{E}_Y, \pi_Y), \vec{y}) \in R^*_{E_0}$, except with negligible probability in the security parameter λ. □

Since games $\boldsymbol{G_2}$ and $\boldsymbol{G_1}$ are equivalent except if event Bad_2 happens, it holds that

$$\Pr[\boldsymbol{G_2} = 1] \leq \Pr[\boldsymbol{G_1} = 1] + \mathsf{negl}(\lambda).$$

```
$\boldsymbol{G_2}$
    $\mathcal{Q} := \emptyset$
    $H := [\bot]$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
    $m^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
    $(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m^*, I_Y)$
    $\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
    if $\mathsf{Adapt}(\hat{\sigma}, \vec{y}) = \sigma^*$
        abort
    $b := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
    return $(m^* \notin \mathcal{Q} \ \wedge \ b)$
```

```
$\mathcal{O}_{\mathrm{S}}(m)$
    $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\sigma$
```

```
$\mathcal{H}(x)$
    if $H[x] = \bot$
        $\mathcal{C} := \{-S+1, \dots, S-1\}$
        $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
    return $H[x]$
```

```
$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$
    Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
    $\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
    if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
        abort
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\hat{\sigma}$
```

Figure 3.3: The formal definition of game $\boldsymbol{G_2}$.

Game $\boldsymbol{G_3}$: This game extends the changes of the previous game to the $\mathcal{O}_{\mathrm{pS}}$ oracle by first creating a valid full signature σ by executing the Sig algorithm, and then converting σ into a valid pre-signature using the extracted witness $\vec{y}$. Furthermore, the game computes the randomnesses $(\hat{E}'_1, \dots, \hat{E}'_{t_S})$ and $(E'_1, \dots, E'_{t_S})$ from σ, and simulates the zero-knowledge proofs $(\pi^{\mathsf{S}}_1, \dots, \pi^{\mathsf{S}}_{t_S})$ using the $\hat{E}'_i$ and E'_i values.

Claim 3. *If NIZK proof system Π_{NIZK} is computationally zero-knowledge, then* $\Pr[\boldsymbol{G_3} = 1] \leq \Pr[\boldsymbol{G_2} = 1] + \mathsf{negl}(\lambda)$.

Proof. First, we observe that except for simulating the zero-knowledge proofs π^{S}_i, the rest of the changes between the games $\boldsymbol{G_2}$ and $\boldsymbol{G_3}$ are only syntactical as it involves rewriting the full signature as a pre-signature. Hence, in order to show indistinguishability between the two games we provide a reduction to the zero-knowledge property of Π_{NIZK}. To this end, we construct a simulator $\mathcal{S}$ that breaks the zero-knowledge property by using the adaptor unforgeability adversary $\mathcal{A}$. $\mathcal{S}$ generates a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ in order to simulate $\mathcal{A}$'s queries to the oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$. When answering $\mathcal{O}_{\mathrm{pS}}$ queries, $\mathcal{S}$ submits to the zero-knowledge challenger of Π_{NIZK} the statement $(E_0, \hat{E}'_i, E^i_Y, E'_i)$, and receives back a proof π^{S}_i, which it returns to $\mathcal{A}$ as the output of the $\mathcal{O}_{\mathrm{pS}}$ query.

If the zero-knowledge challenger of Π_{NIZK} used the honest prover to generate the proof, then the view of $\mathcal{A}$ is equal to that of $\boldsymbol{G_2}$, and if it used the simulator, then the view of

$\boldsymbol{G_3}$

```
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$m^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
$(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m^*, I_Y)$
$\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
$\textbf{if } \mathsf{Adapt}(\hat{\sigma}, \vec{y}) = \sigma^*$
    $\textbf{abort}$
$b := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
$\textbf{return } (m^* \notin \mathcal{Q} \ \wedge \ b)$
```

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

```
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$\textbf{return } \sigma$
```

$\mathcal{H}(x)$

```
$\textbf{if } H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
$\textbf{return } H[x]$
```

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

```
$\text{Parse } I_Y \text{ as } (\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
$\textbf{if } ((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    $\textbf{abort}$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\text{Parse } \sigma \text{ as } (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
$\text{Parse } \mathsf{pk} \text{ as } (E_1, \ldots, E_{S-1})$
$\text{Parse } \vec{E}_Y \text{ as } (E_Y^1, \ldots, E_Y^{t_S})$
$\text{Parse } \vec{y} \text{ as } (y_1, \ldots, y_{t_S})$
$\textbf{for } i \in \{1, \ldots, t_S\} \textbf{ do}$
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
$\textbf{return } (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
```

Figure 3.4: The formal definition of game $\boldsymbol{G_3}$.

$\mathcal{A}$ is equal to that of $\boldsymbol{G_3}$. Therefore, if $\mathcal{A}$ can distinguish between the two games with non-negligible advantage, then $\mathcal{S}$ can break the zero-knowledge property of Π_{NIZK}. By the zero-knowledge property of Π_{NIZK} this happens only with negligible probability, hence, the claim follows. □

Game $\boldsymbol{G_4}$: In this game, upon receiving the challenge message m^* from $\mathcal{A}$, the game creates a full signature σ by executing the Sig algorithm, and transforming the resulting signature into a valid pre-signature in the same way as was done 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

$$\Pr[\boldsymbol{G_4} = 1] \leq \Pr[\boldsymbol{G_3} = 1] + \mathsf{negl}(\lambda).$$

$\boldsymbol{G_4}$

```
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$m^* \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
$(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m^*, I_Y)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^\mathsf{S} \leftarrow \Pi_\mathsf{NIZK}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^\mathsf{S}, \ldots, \pi_{t_S}^\mathsf{S} E'_1, \ldots, E'_{t_S})$
$\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
if $\mathsf{Adapt}(\hat{\sigma}, \vec{y}) = \sigma^*$
    abort
$b := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
return $(m^* \notin \mathcal{Q} \ \wedge \ b)$
```

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

```
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

$\mathcal{H}(x)$

```
if $H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
return $H[x]$
```

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

```
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_\mathsf{NIZK}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^\mathsf{S} \leftarrow \Pi_\mathsf{NIZK}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^\mathsf{S}, \ldots, \pi_{t_S}^\mathsf{S} E'_1, \ldots, E'_{t_S})$
```

Figure 3.5: The formal definition of game $\boldsymbol{G_4}$.

Having shown that the transition from the original aSigForge game (Game $\boldsymbol{G_0}$) to Game $\boldsymbol{G_4}$ is indistinguishable, it remains to show that there exists a simulator (i.e., a reduction) that perfectly simulates $\boldsymbol{G_4}$, and uses $\mathcal{A}$ to win the StrongSigForge game. In the following we describe concisely the simulator code.

```
$\mathcal{S}^{\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}},\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}}(\mathsf{pk})$

$\mathcal{Q} := \emptyset$
$H := [\perp]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$m^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot),\mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
$(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$
$\sigma \leftarrow \mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}(m^*)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
$\sigma^* \leftarrow \mathcal{A}(\hat{\sigma}, I_Y)$
return $(m^*, \sigma^*)$
```

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

$\sigma \leftarrow \mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

```
$\mathcal{H}(x)$

if $H[x] = \perp$
    $H[x] \leftarrow \mathcal{H}^{\mathsf{CSI\text{-}FiSh}}(x)$
return $H[x]$
```

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

Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}(m)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
```

Figure 3.6: The formal definition of the simulator (i.e., reduction).

Simulation of oracle queries. Next, we show how the simulation of the oracle queries are handled.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_{\mathrm{S}}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and returns its response to $\mathcal{A}$.

Random oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}$ on input x, if $H[x] = \perp$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}(x)$, 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 $\vec{y}$ using the extractability of Π_{NIZK}, forwards m to the oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$, and parses the signature that is generated as $\sigma := (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$.
2. $\mathcal{S}$ generates a pre-signature from $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$ by computing $\hat{r}_i \leftarrow r_i - y_i$, for all $i \in \{1, \ldots, t_S\}$.
3. Finally, $\mathcal{S}$ simulates the zero-knowledge proofs π_i^{S}, for $i \in \{1, \ldots, t_S\}$, proving that $\hat{E}_i'$ and E_i' have the same group action. The simulator outputs $\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S}, \pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}}, E_1', \ldots, E_{t_S}')$.

Challenge phase:

1. Upon $\mathcal{A}$ outputting the message m^* as the challenge message, $\mathcal{S}$ generates $(I_Y, \vec{y}) \leftarrow \mathsf{GenR}(1^\lambda)$, forwards m^* to the oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$, and parses the signature that is generated as $\sigma := (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$.
2. $\mathcal{S}$ generates the required pre-signature $\hat{\sigma}$ 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 difference between the simulation and $\boldsymbol{G_4}$ are syntactical. More concretely, instead of generating the secret/public keys and running the algorithms Sig and $\mathcal{H}$, the simulator $\mathcal{S}$ uses its oracles $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$. It remains to show that the forgery output by $\mathcal{A}$ can be used by the simulator (i.e., reduction) to win the $\mathsf{StrongSigForge}$ game.

Claim 4. *(m^*, σ^*) constitutes a valid forgery in the* $\mathsf{StrongSigForge}$ *game.*

Proof. In order to prove this claim, we have to show that the tuple (m^*, σ^*) has not been output by the oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ before. We 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, $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ 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 $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ during the challenge phase only with negligible probability. Therefore, $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ has never output σ^* on query m^* before, and consequently (m^*, σ^*) constitutes a valid forgery for the $\mathsf{StrongSigForge}$ game. □

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \mathsf{negl}(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain

$$\begin{aligned}\mathsf{Adv}_{\mathcal{A}}^{\mathsf{aSigForge}} &= \Pr[\boldsymbol{G_0} = 1] \\ &\leq \Pr[\boldsymbol{G_4}] + \mathsf{negl}(\lambda) \\ &\leq \mathsf{Adv}_{\mathcal{S}}^{\mathsf{StrongSigForge}} + \mathsf{negl}(\lambda).\end{aligned}$$

Since CSI-FiSh is secure in QROM with $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$ modeled as a quantum random oracle, this implies that IAS is $\mathsf{aEUF\text{-}CMA}$ secure in QROM. This concludes the proof of Lemma 3. □

Lemma 4 (Witness Extractability)**.** *Assuming that the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ is* SUF-CMA *secure and $R^*_{E_0}$ is a hard relation, the adaptor signature scheme $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$, as defined in Algorithm 3, is witness extractable.*

Proof. First, we start with the main intuition behind the witness extractability proof. In overall this proof is very similar to the proof of Lemma 3. Our goal is to reduce the witness extractability of $\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}$ to the strong unforgeability of the CSI-FiSh signature scheme. More concretely, assuming that there exists a PPT adversary $\mathcal{A}$ who wins the $\mathsf{aWitExt}$ experiment, we design another PPT adversary $\mathcal{S}$ that wins the $\mathsf{StrongSigForge}$ experiment.

Similar to the unforgeability proof, the main challenge arises during the simulation of pre-sign queries. Hence, the simulation is done exactly as in the proof of Lemma 3. However, unlike in the $\mathsf{aSigForge}$ experiment, in $\mathsf{aWitExt}$, $\mathcal{A}$ outputs the statement I_Y for the relation $R^*_{E_0}$ alongside the challenge message m^*. This means that the game does not choose the pair $(I_Y, \vec{y})$. Therefore, $\mathcal{S}$ does not learn the witness $\vec{y}$, and hence, cannot transform a valid full signature to a pre-signature by computing $\hat{r}_i \leftarrow r_i - y_i$, for all $i \in \{1, \ldots, t_S\}$. Though, it is possible to extract $\vec{y}$ from the zero-knowledge proof embedded in I_Y. After extracting $\vec{y}$, the same approach used in Lemma 3 to simulate the pre-sign queries can be taken here as well.

Next, we continue with the sequence of games needed for the proof.

Game $\boldsymbol{G_0}$: This game corresponds to the original $\mathsf{aWitExt}$ game, where the adversary $\mathcal{A}$ has to come up with a valid signature σ for a message m of its choice, given a pre-signature $\hat{\sigma}$ and a statement/witness pair $(I_Y := (\vec{E}_Y, \pi_Y), \vec{y})$, while having access to oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$, such that $(I_Y, \mathsf{Ext}(\sigma, \hat{\sigma}, I_Y)) \notin R^*_{E_0}$. Since we are in the random oracle model, we explicitly write the random oracle code $\mathcal{H}$. It trivially follows that

$$\Pr[\boldsymbol{G_0} = 1] = \Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R^*_{E_0},\Sigma_{\mathsf{CSI\text{-}FiSh}}}}(\lambda) = 1].$$

Game $\boldsymbol{G_1}$: This game only applies changes to the $\mathcal{O}_{\mathrm{pS}}$ oracle compared to the previous game. More precisely, during the $\mathcal{O}_{\mathrm{pS}}$ queries, this game extracts a witness $\vec{y}$ by executing the extractor algorithm $\mathcal{K}$ on inputs the statement $\vec{E}_Y$, the proof π_Y and the list of random oracle queries H. The game aborts, if for the extracted witness $\vec{y}$ it does not hold that $(I_Y := (\vec{E}_Y, \pi_Y), \vec{y}) \in R^*_{E_0}$.

Claim 5. *Let Bad_1 be the event that $\boldsymbol{G_1}$ aborts during an execution of $\mathcal{O}_{\mathrm{pS}}$, then it holds that* $\Pr[\mathsf{Bad}_1] \leq \mathsf{negl}(\lambda)$*.*

```
$\boldsymbol{G_0}$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\hat{\sigma})$
$\vec{y}' := \mathsf{Ext}(\sigma, \hat{\sigma}, I_Y)$
$b_1 := \mathsf{Ver}(\mathsf{pk}, m, \sigma)$
$b_2 := m \notin \mathcal{Q}$
$b_3 := (I_Y, \vec{y}') \notin R^*_{E_0}$
return $(b_1 \ \wedge \ b_2 \ \wedge \ b_3)$
```

```
$\mathcal{O}_{\mathrm{S}}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

```
$\mathcal{H}(x)$
if $H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow_{\$} \mathcal{C}^{t_S}$
return $H[x]$
```

```
$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\hat{\sigma}$
```

Figure 3.7: The formal definition of game $\boldsymbol{G_0}$.

Proof. According to the *online extractor* property of Π_{NIZK}, for a witness $\vec{y}$ extracted from a proof π_Y for statement $\vec{E}_Y$ such that $\Pi_{\mathsf{NIZK}}.\mathsf{V}(\vec{E}_Y, \pi_Y) = 1$, it holds that $((\vec{E}_Y, \pi_Y), \vec{y}) \in R^*_{E_0}$, except with negligible probability. □

Since games $\boldsymbol{G_1}$ and $\boldsymbol{G_0}$ are equivalent except when the event Bad_1 happens, it holds that

$$\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_1} = 1] + \mathsf{negl}(\lambda).$$

Game $\boldsymbol{G_2}$: This game extends the changes to the $\mathcal{O}_{\mathrm{pS}}$ oracle from the previous game. In the $\mathcal{O}_{\mathrm{pS}}$ execution, this game first creates a valid full signature σ by executing the Sig algorithm, and converts σ into a pre-signature using the extracted witness $\vec{y}$. Moreover, the game computes the randomnesses $(\hat{E}'_1, \ldots, \hat{E}'_{t_S})$ and $(E'_1, \ldots, E'_{t_S})$ from σ, and simulates the zero-knowledge proofs $(\pi^{\mathsf{S}}_1, \ldots, \pi^{\mathsf{S}}_{t_S})$ using the $\hat{E}'_i$ and E'_i values.

As shown in Claim 3, we have that due to the *zero-knowledge* property of Π_{NIZK}, the simulator can generate a proof π^{S}_i that is computationally indistinguishable from an honest proof $\pi_i \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}((E_0, \hat{E}'_i, E_Y, E'_i), b_i)$. Hence, this game is indistinguishable from the previous game, and it holds that

$$\Pr[\boldsymbol{G_1} = 1] \leq \Pr[\boldsymbol{G_2} = 1] + \mathsf{negl}(\lambda).$$

```
$\boldsymbol{G_1}$
    $\mathcal{Q} := \emptyset$
    $H := [\bot]$
    $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
    $(m^*, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m^*, I_Y)$
    $\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\hat{\sigma})$
    $\vec{y}' := \mathsf{Ext}(\sigma^*, \hat{\sigma}, I_Y)$
    $b_1 := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
    $b_2 := m^* \notin \mathcal{Q}$
    $b_3 := ((I_Y, \vec{y}') \notin R^*_{E_0}$
    return $(b_1 \ \wedge \ b_2 \ \wedge \ b_3)$

$\mathcal{O}_\mathrm{S}(m)$
    $\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\sigma$

$\mathcal{H}(x)$
    if $H[x] = \bot$
        $\mathcal{C} := \{-S+1, \ldots, S-1\}$
        $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
    return $H[x]$

$\mathcal{O}_\mathrm{pS}(m, I_Y)$
    Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
    $\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
    if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
        abort
    $\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
    $\mathcal{Q} := \mathcal{Q} \cup \{m\}$
    return $\hat{\sigma}$
```

Figure 3.8: The formal definition of game $\boldsymbol{G_1}$.

Game $\boldsymbol{G_3}$: In this game we apply the exact same changes made in the game $\boldsymbol{G_1}$ for $\mathcal{O}_\mathrm{pS}$ oracle to the challenge phase of the game. During the challenge phase, this game extracts a witness $\vec{y}$ by executing the extractor algorithm $\mathcal{K}$ on inputs the statement $\vec{E}_Y$, the proof π_Y and the list of random oracle queries H. If for the extracted witness y it does not hold that $((\vec{E}_Y, \pi_Y), \vec{y}) \in R^*_{E_0}$, then the game aborts.

Claim 6. *Let* Bad_2 *be the event that* $\boldsymbol{G_3}$ *aborts during the challenge phase, then it holds that* $\Pr[\mathsf{Bad}_2] \leq \mathsf{negl}(\lambda)$.

Proof. This proof is analogous to the proof of $\boldsymbol{G_1}$ in the proof of Lemma 4. □

Since games $\boldsymbol{G_2}$ and $\boldsymbol{G_3}$ are identical except if event Bad_2 occurs, it holds that

$$\Pr[\boldsymbol{G_2} = 1] \leq \Pr[\boldsymbol{G_3} = 1] + \mathsf{negl}(\lambda).$$

Game $\boldsymbol{G_4}$: In this game we apply the exact same changes made in the game $\boldsymbol{G_2}$ for $\mathcal{O}_\mathrm{pS}$ oracle to the challenge phase of the game. In the challenge phase, this game first creates a valid full signature σ by executing the Sig algorithm, and converts σ into a pre-signature using the extracted witness $\vec{y}$. Moreover, the game computes the randomness and zero-knowledge proofs as described in the game $\boldsymbol{G_2}$, and hence, the same arguments

```
$\boldsymbol{G_2}$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m^*, I_Y)$
$\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\hat{\sigma})$
$\vec{y}' := \mathsf{Ext}(\sigma^*, \hat{\sigma}, I_Y)$
$b_1 := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
$b_2 := m^* \notin \mathcal{Q}$
$b_3 := ((I_Y, \vec{y}') \notin R^*_{E_0}$
**return** $(b_1 \ \wedge \ b_2 \ \wedge \ b_3)$
```

```
$\mathcal{O}_\mathrm{S}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
**return** $\sigma$
```

```
$\mathcal{H}(x)$
**if** $H[x] = \bot$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
**return** $H[x]$
```

```
$\mathcal{O}_\mathrm{pS}(m, I_Y)$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_\mathsf{NIZK}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
**if** $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    **abort**
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
**for** $i \in \{1, \ldots, t_S\}$ **do**
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^\mathsf{S} \leftarrow \Pi_\mathsf{NIZK}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
**return** $(\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
            $\pi_1^\mathsf{S}, \ldots, \pi_{t_S}^\mathsf{S} E'_1, \ldots, E'_{t_S})$
```

Figure 3.9: The formal definition of game $\boldsymbol{G_2}$.

also apply here. Therefore, this game is indistinguishable from the previous game, and it holds that

$$\Pr[\boldsymbol{G_3} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \mathsf{negl}(\lambda).$$

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 (i.e., reduction) that perfectly simulates $\boldsymbol{G_4}$, and uses $\mathcal{A}$ to win the StrongSigForge game. In the following we describe concisely the simulator code.

Simulation of oracle queries. Next, we show how the simulation of the oracle queries are handled.

$\boldsymbol{G_3}$

```
$\mathcal{Q} := \emptyset$
$H := [\perp]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\hat{\sigma} \leftarrow \mathsf{PreSig}(\mathsf{sk}, m, I_Y)$
$\sigma \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\hat{\sigma})$
$\vec{y}' := \mathsf{Ext}(\sigma^*, \hat{\sigma}, I_Y)$
$b_1 := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
$b_2 := m^* \notin \mathcal{Q}$
$b_3 := ((E_Y, \pi_Y), \vec{y}') \notin R^*_{E_0}$
return $(b_1 \ \wedge \ b_2 \ \wedge \ b_3)$
```

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

```
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

$\mathcal{H}(x)$

```
if $H[x] = \perp$
    $\mathcal{C} := \{-S+1, \ldots, S-1\}$
    $H[x] \leftarrow\!\!\$\ \mathcal{C}^{t_S}$
return $H[x]$
```

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

```
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i), 1)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
            $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
```

Figure 3.10: The formal definition of game $\boldsymbol{G_3}$.

Signing queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{O}_\mathrm{S}$ on input m, $\mathcal{S}$ forwards m to its oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and returns its response to $\mathcal{A}$.

Random oracle queries: Upon $\mathcal{A}$ querying the oracle $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$ on input x, if $H[x] = \perp$, then $\mathcal{S}$ queries $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}(x)$, 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 $\vec{y}$ using the extractability of Π_{NIZK}, forwards m to oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and parses the signature that is generated as $\sigma := (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$.
2. $\mathcal{S}$ generates a pre-signature from $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$ by computing $\hat{r}_i \leftarrow r_i - y_i$, for all $i \in \{1, \ldots, t_S\}$.

```
$\boldsymbol{G_4}$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\mathsf{pk})$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_\mathsf{NIZK}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
Parse $\sigma$ as $(r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \dots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \dots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \dots, y_{t_S})$
for $i \in \{1, \dots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^\mathsf{S} \leftarrow \Pi_\mathsf{NIZK}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\hat{\sigma} := (\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S},$
        $\pi_1^\mathsf{S}, \dots, \pi_{t_S}^\mathsf{S} E'_1, \dots, E'_{t_S})$
$\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_\mathrm{S}(\cdot), \mathcal{O}_\mathrm{pS}(\cdot,\cdot)}(\hat{\sigma})$
$\vec{y}' := \mathsf{Ext}(\sigma^*, \hat{\sigma}, I_Y)$
$b_1 := \mathsf{Ver}(\mathsf{pk}, m^*, \sigma^*)$
$b_2 := m^* \notin \mathcal{Q}$
$b_3 := ((E_Y, \pi_Y), \vec{y}') \notin R^*_{E_0}$
return $(b_1 \wedge b_2 \wedge b_3)$
```

```
$\mathcal{O}_\mathrm{S}(m)$
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$
```

```
$\mathcal{H}(x)$
if $H[x] = \bot$
    $\mathcal{C} := \{-S+1, \dots, S-1\}$
    $H[x] \leftarrow\$ \mathcal{C}^{t_S}$
return $H[x]$
```

```
$\mathcal{O}_\mathrm{pS}(m, I_Y)$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_\mathsf{NIZK}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}(\mathsf{sk}, m)$
Parse $\sigma$ as $(r_1, \dots, r_{t_S}, c_1, \dots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \dots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \dots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \dots, y_{t_S})$
for $i \in \{1, \dots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^\mathsf{S} \leftarrow \Pi_\mathsf{NIZK}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S},$
        $\pi_1^\mathsf{S}, \dots, \pi_{t_S}^\mathsf{S} E'_1, \dots, E'_{t_S})$
```

Figure 3.11: The formal definition of game $\boldsymbol{G_4}$.

3. Finally, $\mathcal{S}$ simulates the zero-knowledge proofs π_i^S, for $i \in \{1, \dots, t_S\}$, proving that $\hat{E}'_i$ and E'_i have the same group action. The simulator outputs $\hat{\sigma} := (\hat{r}_1, \dots, \hat{r}_{t_S}, c_1, \dots, c_{t_S}, \pi_1^\mathsf{S}, \dots, \pi_{t_S}^\mathsf{S}, E'_1, \dots, E'_{t_S})$.

```
$\mathcal{S}^{\mathsf{Sig}^{\textsf{CSI-FiSh}}, \mathcal{H}^{\textsf{CSI-FiSh}}}(\mathsf{pk})$
$\mathcal{Q} := \emptyset$
$H := [\bot]$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, I_Y) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\mathsf{pk})$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}^{\textsf{CSI-FiSh}}(m^*)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\hat{\sigma} := (\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
$\sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}}(\cdot), \mathcal{O}_{\mathrm{pS}}(\cdot,\cdot)}(\hat{\sigma})$
return $(m^*, \sigma^*)$

$\mathcal{O}_{\mathrm{S}}(m)$
$\sigma \leftarrow \mathsf{Sig}^{\textsf{CSI-FiSh}}(m)$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $\sigma$

$\mathcal{H}(x)$
if $H[x] = \bot$
    $H[x] \leftarrow \mathcal{H}^{\textsf{CSI-FiSh}}(x)$
return $H[x]$

$\mathcal{O}_{\mathrm{pS}}(m, I_Y)$
Parse $I_Y$ as $(\vec{E}_Y, \pi_Y)$
$\vec{y} := \Pi_{\mathsf{NIZK}}.\mathcal{K}(\vec{E}_Y, \pi_Y, H)$
if $((\vec{E}_Y, \pi_Y), \vec{y}) \notin R^*_{E_0}$
    abort
$\sigma \leftarrow \mathsf{Sig}^{\textsf{CSI-FiSh}}(m)$
Parse $\sigma$ as $(r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$
Parse $\mathsf{pk}$ as $(E_1, \ldots, E_{S-1})$
Parse $\vec{E}_Y$ as $(E_Y^1, \ldots, E_Y^{t_S})$
Parse $\vec{y}$ as $(y_1, \ldots, y_{t_S})$
for $i \in \{1, \ldots, t_S\}$ do
    $\hat{r}_i \leftarrow r_i - y_i$
    $\hat{E}'_i \leftarrow [r_i]E_{c_i}$
    $E'_i \leftarrow [y_i]\hat{E}'_i$
    $\pi_i^{\mathsf{S}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathcal{S}((E_0, \hat{E}'_i, E_Y^i, E'_i))$
$\mathcal{Q} := \mathcal{Q} \cup \{m\}$
return $(\hat{r}_1, \ldots, \hat{r}_{t_S}, c_1, \ldots, c_{t_S},$
        $\pi_1^{\mathsf{S}}, \ldots, \pi_{t_S}^{\mathsf{S}} E'_1, \ldots, E'_{t_S})$
```

Figure 3.12: The formal definition of the simulator (i.e., reduction).

Challenge phase:

1. Upon $\mathcal{A}$ outputting the message (m^*, I_Y) as the challenge message, $\mathcal{S}$ extracts $\vec{y}$ using the extractability of Π_{NIZK}, forwards m^* to the oracle $\mathsf{Sig}^{\textsf{CSI-FiSh}}$ and parses the signature that is generated as $\sigma := (r_1, \ldots, r_{t_S}, c_1, \ldots, c_{t_S})$.
2. $\mathcal{S}$ generates the required pre-signature $\hat{\sigma}$ 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 difference between the simulation and $\boldsymbol{G_4}$ are syntactical. More precisely, instead of generating the secret/public keys and running the algorithms Sig and $\mathcal{H}$, the simulator $\mathcal{S}$ uses its oracles $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ and $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$. It remains to show that the signature output by $\mathcal{A}$ can be used by the simulator to win the $\mathsf{StrongSigForge}$ game.

Claim 7. *(m^*, σ^*) constitutes a valid forgery in the* $\mathsf{StrongSigForge}$ *game.*

Proof. In order to prove this claim, we have to show that the tuple (m^*, σ^*) has not been output by the oracle $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ before. We 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, $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ 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 $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ during the challenge phase, the extracted $\vec{y}$ would be in relation with the given public statement I_Y. Therefore, $\mathsf{Sig}^{\mathsf{CSI\text{-}FiSh}}$ has never output σ^* on query m^* before and consequently (m^*, σ^*) constitutes a valid forgery for the $\mathsf{StrongSigForge}$ game. □

From the games $\boldsymbol{G_0} - \boldsymbol{G_4}$ we get that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \mathsf{negl}(\lambda)$. Since $\mathcal{S}$ provides a perfect simulation of game $\boldsymbol{G_4}$, we obtain

$$\begin{aligned} \mathsf{Adv}_{\mathsf{aWitExt}} &= \Pr[\boldsymbol{G_0} = 1] \\ &\leq \Pr[\boldsymbol{G_4} = 1]\mathsf{negl}(\lambda) \\ &\leq \mathsf{Adv}_{\mathcal{S}}^{\mathsf{StrongSigForge}} + \mathsf{negl}(\lambda). \end{aligned}$$

Since CSI-FiSh is secure in QROM with $\mathcal{H}^{\mathsf{CSI\text{-}FiSh}}$ modeled as a quantum random oracle, this implies that IAS achieves witness extractability even against quantum adversaries. This concludes the proof of Lemma 4. □

This concludes the proof of Theorem 1. □

Next, we continue with the proof of the optimized variant of IAS ($\mathsf{O\text{-}IAS}$), as defined in Section 3.2. More concretely, we prove the following theorem.

Theorem 2. *Let Π_{NIZK} be a NIZKPoK with an online extractor, the CSI-FiSh signature scheme $\Sigma_{\mathsf{CSI\text{-}FiSh}}$ be* $\mathsf{SUF\text{-}CMA}$ *secure and $R^{\dagger}_{E_0}$ br a hard relation, the adaptor signature scheme $\Xi_{R^{\dagger}_{E_0}, \Sigma_{\mathsf{CSI\text{-}FiSh}}}$, is secure in QROM.*

Proof. The proof is analogous to the proof of Theorem 1, with the only changes being to the underlying hard relation and the adaptation procedure. More concretely, the hard relation $R^*_{E_0}$, which consists of pairs $I_Y := (\vec{E}_Y, \pi_Y)$, such that $\vec{E}_Y \in L_{R^{t_S}_{E_0}}$ and π_Y is a zero-knowledge proof that $\vec{E}_Y \in L_{R^{t_S}_{E_0}}$, is replaced with the hard relation $R^\dagger_{E_0}$, which consists of pairs $I_Y := (E_Y, \pi_Y)$, such that $E_Y \in L_{R^1_{E_0}}$ and π_Y is a zero-knowledge proof that $E_Y \in L_{R^1_{E_0}}$. Since both $R^*_{E_0}$ and $R^\dagger_{E_0}$ are hard relations, this proof goes through exactly as the proof of Theorem 1, with the only caveat being that in the O-IAS construction solely the first iteration of the pre-signature needs to be adapted. Hence, when the simulator $\mathcal{S}$ transforms a full signature into a pre-signature for the adversary $\mathcal{A}$, it only needs to modify the first part of the signature (i.e., r_1). This concludes the proof of Theorem 2. □

3.4 Performance Evaluation

In order to evaluate IAS we extended the commit `7a9d30a` version of the proof-of-concept implementation of CSI-FiSh[4]. The implementation depends on the eXtended Keccak Code Package for the implementation of SHAKE256, which is used as the underlying hash function and to expand the randomness. We also use the GMP library [Gt19] for high precision arithmetic. We implemented the optimized variant O-IAS as explained in Section 3.2. Since O-IAS and CSI-FiSh are composed of multiple independent iterations of a non-interactive identification scheme, they are amenable to parallelization. Hence, we provided a parallelized implementation using OpenMP. The source code is available at https://github.com/etairi/Adaptor-CSI-FiSh.

Parameters. CSI-FiSh signature scheme is instantiated with the following parameters: i) S, the number of public keys to use, ii) t_S, the number of repetitions to perform, and iii) k, the rate of the slow hash function (e.g., $k = 16$ means that the used hash function is a factor 2^{16} slower than a standard hash function, such as SHA-3). In order to ensure λ bits of soundness security it suffices to take the parameters such that $S^{-t_S} \leq 2^{-\lambda+k}$. As is described in [BKV19], the parameter S controls the trade-off between on the one hand small public key and fast key generation (when S is small), and on the other hand small signature and fast signing/verification (when S is large).

Testbed. All benchmarks were taken on a KVM-based VM with 2.0GHz AMD EPYC 7702 processor with 16 cores and 32GB RAM, running Ubuntu 18.04 LTS, and the code was compiled with gcc 7.5.0.

3.4.1 Evaluation Results

In this section, we present our evaluation results and discuss the communication size and computation time of O-IAS (i.e., sizes of objects and running times of the algorithms). The results of our evaluation are summarized in Table 3.1. As shown, playing with the parameters we can obtain different trade-offs, which we explain next. We divide our

[4] https://github.com/KULeuven-COSIC/CSI-FiSh

discussion on: (i) on-chain analysis (i.e., overhead imposed on the blockchain to support O-IAS) and (ii) off-chain analysis (i.e., overhead for peers at the application level).

On-Chain Analysis. In order to support O-IAS, the blockchain only needs to verify that each transaction is accompanied by a signature that correctly verifies under a given public key according to the logic of the verification algorithm of CSI-FiSh. Thus, the *storage size* imposed by CSI-FiSh is dominated by the signature and public key sizes and the goal is thus to minimize these values. As was already described above, the parameter S can be set to a small value to achieve compact public keys. This, however, yields larger signatures. For instance, we can observe from Table 3.1 that by setting $S = 2$ one can have public keys of only 128 bytes, but at the cost of signatures of size 1880 bytes.

Similarly, the *computation time* of IAS for the miners is represented by the running time of the verification algorithm of CSI-FiSh. In our evaluation, we observe that increasing the value of S reduces the verification time of CSI-FiSh. However, as was already noted, this increases the public key sizes. Nevertheless, the technique of using Merkle trees to obtain compact and constant size public keys (but large secret keys) as described in [BKV19] can also be applied to our construction. Using that technique one can have public keys of size 32 bytes, signatures of size 1995 bytes and verification algorithm running time of 370 milliseconds with no parallelization, as shown in [BKV19, Table 4], or 60 milliseconds with our parallelized implementation.

Off-Chain Analysis. The operations of O-IAS defined in Algorithm 3 are carried out off-chain, meaning that the creation and verification of pre-signatures is done in a peer-to-peer manner and thus do not need to be stored in the blockchain, nor to be verified by the miners. Yet, we discuss here the computation time and communication size for this part as it illustrates the overhead for applications building upon O-IAS.

In terms of *communication size*, a pre-signature $\hat{\sigma}$ in IAS has size of $\sim$ 19KB on average. We can observe from Table 3.1 that the pre-signature size only varies slightly the change in parameters. The reason for this is that the pre-signature size is dominated by the expensive zero-knowledge proof for L_2 (see Section 3.1.1) that is required during pre-signature computation, which has size $\sim$ 18KB, and it varies slightly with parameter k (bigger k implies smaller proof size). On the other hand the running times of the

S	t_S	k	$\lvert\mathsf{sk}\rvert$	$\lvert\mathsf{pk}\rvert$	$\lvert\hat{\sigma}\rvert$	$\lvert\sigma\rvert$	KeyGen	Sig	Ver	PreSig	PreVer	Ext	Adapt
2^1	56	16	16	128	19944	1880	0.05	0.24	0.23	3.59	3.55	0.005	0.005
2^2	38	14	16	256	19672	1286	0.06	0.16	0.16	2.75	2.68	0.005	0.005
2^3	28	16	16	512	19020	956	0.07	0.13	0.14	2.21	2.15	0.005	0.005
2^4	23	13	16	1024	19338	791	0.07	0.11	0.11	1.99	1.94	0.005	0.005
2^6	16	16	16	4096	18624	560	0.29	0.08	0.09	1.61	1.56	0.005	0.005
2^8	13	11	16	16384	19330	461	1.00	0.08	0.08	1.50	1.44	0.005	0.005
2^{10}	11	7	16	65536	19908	395	3.21	0.06	0.06	1.40	1.36	0.005	0.005
2^{12}	9	11	16	262144	19198	329	12.89	0.06	0.06	1.30	1.25	0.005	0.005
2^{15}	7	16	16	2097152	18327	263	102.02	0.06	0.06	1.16	1.11	0.005	0.005

Table 3.1: Performance of O-IAS. Time is shown in seconds and size in bytes.

pre-signature and pre-verification algorithms decrease with the increased S value, meaning with the decreased number of iterations t_S. The reason for this is that during pre-signature and pre-verification computation our implementation only parallelizes the computation of the zero-knowledge proof for L_2, but all the t_S iterations are computed by a single thread. We opted for this approach as the zero-knowledge proof is the dominating cost in IAS, and it requires $\sim$ 750 milliseconds to compute and verify. On the other hand, extraction and adaptation are generally extremely fast operations for our construction, however, we point out that the time for extraction in Table 3.1 does not include the verification that the extracted witness $\vec{y}$, which is a vector of size 1 for O-IAS, satisfies $(I_Y, \vec{y}) \in R^*_{E_0}$ (line 49 in Algorithm 3). We note that in practice one can just extract the witness, adapt the pre-signature and then check that the signature verifies, which is more efficient than actually checking in $R^*_{E_0}$, which requires verifying an expensive zero-knowledge proof.

Lastly, we note that even though the communication size is a bit high these operations are handled off-chain, and the pre-signatures are not stored in the blockchain.

3.4.2 Comparison with LAS

We compare our evaluation results with those of LAS [EEE20], which is the only other known post-quantum AS, regarding on-chain and off-chain overhead. The authors of [EEE20] did not provide any implementation, but they estimated the size of their signature and pre-signature as 2701 and 3210 bytes, respectively. From this we can observe that our signature sizes are $1.5 - 10$x smaller depending on the parameter choices, however, our pre-signature sizes are $\sim$ 6x larger. Though, due to the *weak* pre- signature adaptability property of LAS the applications that use LAS require an expensive zero-knowledge proof to ensure that the extracted witness is of correct norm. In [LNP22] it is shown that such a proof has size of more than 10KB, which signifies that our construction is more efficient with respect to both on-chain and off-chain communication size. Moreover, LAS has public key size of 1472 bytes (observed from [DLL$^+$17, Table 2]), which implies that using the Merkle tree technique we can have public key sizes that are 42x times smaller. In terms of computation time, LAS is an AS scheme based on Dilithium [DLL$^+$17], and thus, it can perform more than a hundred sign/verify operations per second, as these operations take less than one millisecond for Dilithium, thereby offering better computational performance than O-IAS.

In summary, our evaluation shows that it is feasible to adopt IAS to extend current blockchains with post-quantum AS at the cost of $\sim$ 1500 bytes (for combined public key and signature size using parameters $S = 2^3, t_S = 28, k = 16$) of communication size, which will be $\sim$ 3x smaller than LAS, and requiring only $\sim$ 100 milliseconds of computation time (for signature verification using the same parameters). Analogous results and reduction in communication size also applies to the off-chain setting, which greatly benefits the off-chain applications using AS as building block, such as payment channels, payment-channel networks, atomic swaps or payment-channel hubs, which are performed over a WAN network, and thus, a reduction in communication is desirable.

CHAPTER 4

Universally Composable Adaptor Signatures

In this chapter we present universally composable adaptor signatures. First, in Section 4.1 we abstract away the conditions used within adaptor signatures in a standalone functionality $\mathcal{G}_{\mathsf{Cond}}$. Then, in Section 4.2 we model the adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ and prove its realization with the help of $\mathcal{G}_{\mathsf{Cond}}$. We note that in this chapter and in the rest of the thesis we consider two-party adaptor signatures with aggregatable keys (as defined in Section 2.2.6).

4.1 Global Conditions

In existing blockchain protocols, adaptor signatures and their associated conditions are analyzed within monolithic protocol descriptions. Here, we consider a more modular handling and analyzes of conditions, by abstracting them away using a standalone global functionality $\mathcal{G}_{\mathsf{Cond}}$.

Intuitively, a cryptographic condition describes the properties as given by a hard relation R (as described in Section 2.2.1). Concretely, a condition is identified by a public statement and we say that the condition is satisfied if the corresponding witness is provided. Due to the hardness of the relation, without prior knowledge, it is hard to come up with a witness satisfying a given condition. A typical example is the discrete logarithm (DL) assumption over certain cyclic groups $(\mathbb{G}, g, q)$ (with generator g and order q), where given a group element (the statement) $Y = g^y$, it is hard to compute the exponent y (the witness).

At first sight, it may seem counter-intuitive to define conditions as a standalone ideal functionality. The reason for doing so is that the prerequisites for a party to craft a witness related to a statement often emerge from a cryptographic protocol for the condition creation. As an example, consider the following scenario. Alice plays a simple guessing game with Bob. If Bob can guess a number between 1 and 10 then he gets a prize from Alice. To implement this based on the DL assumption, Alice prepares ten secret witnesses $(y_i^{mask})_{i\in(1,10)}$ and sends the corresponding statements $\overrightarrow{Y^{mask}} = (g^{y_i^{mask}})_{i\in(1,10)}$ to Bob. Bob himself prepares one witness y^{win} and ten witnesses $(y_i^{blank})_{i\in(1,10)}$. For the guess j, Bob prepares $\overrightarrow{Y^{guess}} = (Y_i^{guess})_{i\in(1,10)}$ such that for $i \neq j$, $Y_i^{guess} = (Y_i^{mask})^{y_i^{blank}}$ and $Y_j^{guess} = g^{y^{win}}$. At this point, Bob knows the witness for Y_j^{guess}, while he cannot know the witness for any statement Y_i^{guess} with $i \neq j$, since for this, Bob would require Alice's secret masking values $(y_i^{mask})_{i\in(1,10)}$. Bob proves in zero-knowledge to Alice that $\overrightarrow{Y^{guess}}$ is well-formed. Now, Alice chooses a number m and prepares a payment to Bob based on Y_m^{guess}. Alice at this point, cannot know which condition Bob can open, only that Bob can open exactly one out of the ten provided conditions. If Alice and Bob chose the same number ($j = m$), Bob can complete the payment, otherwise, the money stays with Alice.

For reasoning about the aforedescribed guessing game, we need to capture that there are ten conditions out of which Bob can open exactly one. However, the creation of conditions with this property involves a protocol itself. This condition-creation protocol is independent of more advanced protocols relying on conditions with the respective property.

In summary, by modeling conditions as a separate functionality, we can modularize reasoning about condition creation (e.g., $\overrightarrow{Y^{guess}}$) and protocols using these conditions (e.g., the payment from Alice to Bob based on Y_m^{guess}).

Technically, this means that we can extend the functionality $\mathcal{G}_{\mathsf{Cond}}$ with further types of conditions without reproving any of the results from Chapters 5 and 6. In this thesis, we present only three different forms of conditions:

1. Plain conditions, named as *individual conditions*, which parties can create on their own by creating a fresh witness and the corresponding statement for a given hard relation.

2. Composed conditions, named *merged conditions*, which combine two existing conditions. The concrete composition operation depends on the underlying hard relation and is specified as a parameter f_{merge} to the functionality $\mathcal{G}_{\mathsf{Cond}}$.

3. 1-out-of-n conditions, which enable a party P together with a set of other parties $\mathcal{P}$ to jointly create a vector of statements, such that P (without the collaboration of the parties in $\mathcal{P}$) only knows the witness to exactly one out of these statements. For example, the previously described guessing game included the creation of a 1-out-of-10 condition for the discrete logarithm relation with P = Bob and $\mathcal{P} = \{\text{Alice}\}$.

4.1.1 Ideal Functionality $\mathcal{G}_{\mathsf{Cond}}$

We illustrate the (global) ideal functionality $\mathcal{G}_{\mathsf{Cond}}$ for conditions in Figure 4.1. $\mathcal{G}_{\mathsf{Cond}}$ is parameterized by a hard relation R and a merging function f_{merge}. This is needed to enable composability with the adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ (given in Section 4.2), which is also defined w.r.t. to a hard relation R.

$\mathcal{G}_{\mathsf{Cond}}$ provides three interfaces and maintains a list $\mathcal{L}$ of conditions and their corresponding openings. The individual conditions interface acts as a bulletin board for conditions created outside the ideal functionality, and just stores the input condition/opening (i.e., statement/witness) pair in the list $\mathcal{L}$. The merged conditions interface models the creation of a condition as the composition of two other conditions, where the concrete composition operations are given by f_{merge} function. This function is split into an operation $+$ on witnesses and an operation $\cdot$ on statements, which need to satisfy that $(Y_1 \cdot Y_2, y_1 + y_2) \in R$ if both $(Y_1, y_1), (Y_2, y_2) \in R$. The 1-out-of-$n$ interface models the creation of joint conditions amongst the set of parties, such that the initiating party only knows opening to one of these conditions (which is specified with *index*). Lastly, the open condition interface allows checking if a condition/opening pair is valid (i.e., is in $\mathcal{L}$).

4.1.2 Protocol Π_{Cond}

We describe the global conditions protocol Π_{Cond} in Figure 4.2 for discrete logarithm (DL) relation. More precisely, the protocol is parameterized with a group description $(\mathbb{G}, g, q)$, where g is the generator and q is the order of the group, along with the DL relation R_{DL} over it, i.e., $(Y, y) \in R_{\mathsf{DL}} \iff Y = g^y$. Naturally, we assume that the group $\mathbb{G}$ is a DL-hard group here. The function f_{merge} defines the witness operation $(+)$ as scalar addition and the statement operation $(\cdot)$ as the group operation.

In the case of individual conditions, the protocol checks if the input condition/opening pair (i.e., statement/witness pair for R_{DL}) is valid, and in such a case, returns it. In the case of merged conditions, the protocol multiplies the inputted condition to form the merged condition, which gets returned by this process. For the 1-out-of-n conditions, we propose a multi-party protocol where each party provides a random share s_i for each of the conditions Y_j. Finally, the invoking party combines the shares from other users to compute each Y_j except for the position that she commits to, denoted by *index*, where she just uses the condition for which she knows the opening. Moreover, she also proves in zero-knowledge that the final (set of) conditions are computed correctly. Lastly, for opening conditions, the protocol validates the membership of input condition/opening (i.e., statement/witness) pair in the relation R_{DL}, and returns the output bit b.

4.1.3 Security Proof

The security of our construction is established with the following theorem.

Ideal Functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$

The functionality interacts with an adversary $\mathcal{S}$ and set of parties $\mathcal{P} = \{P_1, \dots, P_n\}$. Additionally, the functionality maintains a list $\mathcal{L}$ that is indexed by conditions (i.e., statements) and stores their corresponding openings (i.e., witnesses). The functionality is parameterized by a hard relation R and a function f_{merge} for which the following invariant holds,

$$(Y_1, y_1) \in R \wedge (Y_2, y_2) \in R \implies (f_{\mathsf{merge}}(stmt, R, (Y_1, Y_2)), f_{\mathsf{merge}}(wit, R, y_1, y_2)) \in R.$$

Individual Conditions: Upon receiving $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y))$ from some party P,

- If $(Y, y) \notin R$, then ignore the request. Otherwise, continue.
- Set $\mathcal{L}[Y] := y$
- Send $(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y)$ to P and $\mathcal{S}$.

Merged Conditions: Upon receiving $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_1, Y_2))$ from some party P,

- If $\mathcal{L}[Y_1] = \perp$ or $\mathcal{L}[Y_2] = \perp$, then ignore the request. Otherwise, continue.
- Set $Y^* := f_{\mathsf{merge}}(stmt, R, (Y_1, Y_2))$, $y^* := f_{\mathsf{merge}}(wit, R, (\mathcal{L}[Y_1], \mathcal{L}[Y_2]))$, and $\mathcal{L}[Y^*] := y^*$.
- Send $(\mathsf{created\text{-}merged\text{-}cond}, \mathsf{sid}, Y^*)$ to P and $\mathcal{S}$.

1-out-of-n Conditions: Upon receiving $(\mathsf{create\text{-}1\text{-}of\text{-}n\text{-}cond}, \mathsf{sid}, (Y, y), index, n, \{P_i\})$ from some party P, do the following:

- If $(Y, y) \notin R$, then ignore the request. Otherwise, continue.
- For all $i \in [n] \wedge i \neq index$, sample random $(Y_i, y_i) \in R$.
- Set $Y^* := (Y_1, \dots, Y_{index} := Y, \dots, Y_n)$.
- For all $P^* \in \{P_i\}$, send $(\mathsf{join\text{-}1\text{-}of\text{-}n\text{-}cond}, \mathsf{sid}, P, Y^*)$, and receive back $(\mathsf{joined\text{-}1\text{-}of\text{-}n\text{-}cond}, \mathsf{sid}, b_i)$.
- If any $b_i = 0$, then abort. Otherwise, continue.
- Set $\mathcal{L}[Y^*] = (\perp, \dots, y_{index} := y, \dots, \perp)$.
- Send $(\mathsf{created\text{-}1\text{-}of\text{-}n\text{-}cond}, \mathsf{sid}, Y^*)$ to P and $\mathcal{S}$.

Open Conditions: Upon receiving $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*))$ from some party P^*,

- Set $b := (\mathcal{L}[Y^*] \overset{?}{=} y^*)$.
- Send $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b)$ to P^* and $\mathcal{S}$.

Figure 4.1: Ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$.

Theorem 3. *Let Π_{NIZK} be a UC-secure non-interactive zero-knowledge proof system and $\mathbb{G}$ be a* DL*-hard group, then the protocol $\Pi_{\mathsf{Cond}}^{R_{\mathsf{DL}}}$ UC-realizes the ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$, for $R = R_{\mathsf{DL}}$ and f_{merge} as defined in Figure 4.2.*

Proof. Throughout the following proof, we implicitly assume that all messages of the adversary are well-formed, and we treat the malformed messages as aborts. We consider a static corruption model, and we denote the set of users corrupted by the adversary with $\mathcal{C}$. The proof is composed of a series of hybrid arguments.

Hybrid $\mathcal{H}_0$: This corresponds to the original Π_{Cond} protocol.

Hybrid $\mathcal{H}_1$: Replace the honestly computed NIZK proof π_{index} with a simulated proof (in 1-out-of-n conditions).

Protocol $\Pi^{R_{\mathsf{DL}}}_{\mathsf{Cond}}$

The protocol is parameterized by group description $(\mathbb{G}, g, q)$, and the corresponding discrete logarithm (DL) relation R_{DL} over it, i.e., $(Y, y) \in R_{\mathsf{DL}} \iff Y = g^y$.

<u>**Individual Conditions:**</u> Party P upon receiving $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y))$ from $\mathcal{E}$, checks if $(Y, y) \in R_{\mathsf{DL}}$. If not, then ignores the request. Otherwise, returns (Y, y).

<u>**Merged Conditions:**</u> Party P upon receiving $(\mathsf{create\text{-}and\text{-}cond}, \mathsf{sid}, (Y_1, Y_2))$ from $\mathcal{E}$, compute $Y^* := Y_1 \cdot Y_2$ and return Y^*.

<u>**Open Conditions:**</u> Party P upon receiving $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*))$ from $\mathcal{E}$, return $((Y^*, y^*) \overset{?}{\in} R_{\mathsf{DL}})$.

<u>**1-out-of-n Conditions:**</u> Party P upon receiving $(\mathsf{create\text{-}1\text{-}of\text{-}n\text{-}cond}, (Y, y), \mathit{index}, n, \{P_i\})$ from $\mathcal{E}$, for all $P^* \in \{P_i\}$, sends $(n, \{P_i\})$ to P^*.

Party $P^* \in \{P_i\}$ upon receiving $(n, \{P_i\})$ from P, does the following:

- For all $j \in [n]$, sample $s_i[j] \leftarrow\$ \mathbb{Z}_q$.
- For all $j \in [n]$, compute $h_i[j] := g^{s[j]}$.
- Send vector $\vec{h}_i := \{h_i[j]\}_{j \in [n]}$ to P and $\{P_i\} \setminus \{P^*\}$.

Party P upon receiving $\vec{h}_i$ from $P^* \in \{P_i\}$, does the following:

- For all $j \in [n]$, sample $s[j] \leftarrow\$ \mathbb{Z}_q$.
- For all $j \in [n]$ and $j \neq \mathit{index}$, compute $f_j := \prod_{i \in |\{P_i\}|} h_i[j]$ and $c[j] := f_j \cdot g^{s[j]}$.
- Set $c[\mathit{index}] := Y$ and $s[\mathit{index}] := y$.
- Compute $\pi_{\mathit{index}} \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}(\{\exists(\vec{s}, \mathit{index}) \mid c[\mathit{index}] = g^{s[\mathit{index}]} \wedge (\forall j \in [n] \ \wedge \ j \neq \mathit{index}, c[j] = f_j \cdot g^{s[j]})\}, (\vec{s}, \mathit{index}))$.
- Return $((\vec{c}, \{f_j\}_{j \in [n]}, \pi_{\mathit{index}}), y)$.

<u>**Definition of f_{merge}:**</u>

- $f_{\mathsf{merge}}(\mathit{stmt}, R, (Y_1, Y_2)) := Y_1 \cdot Y_2$
- $f_{\mathsf{merge}}(\mathit{wit}, R, (y_1, y_2)) := y_1 + y_2$

Figure 4.2: Protocol $\Pi^{R_{\mathsf{DL}}}_{\mathsf{Cond}}$.

<u>Hybrid $\mathcal{H}_2$:</u> For the set of corrupted parties $\mathcal{C}$, if the adversary outputs $(\mathsf{open\text{-}cond}, (Y^*, y^*))$, such that $(Y^*, y^*) \in R_{\mathsf{DL}}$, for the condition Y^* created by some party P via $(\mathsf{create\text{-}ind\text{-}cond}, (Y^*, y^*))$ and $P \notin \mathcal{C}$ (i.e., uncorrupted party), then the experiment aborts by outputting fail_1.

<u>Hybrid $\mathcal{H}_3$:</u> For the set of corrupted parties $\mathcal{C}$, if the adversary outputs $(\mathsf{open\text{-}cond}, (\vec{c}^*[\mathit{index}^*], y^*))$, such that $(\vec{c}^*[\mathit{index}^*], y^*) \in R_{\mathsf{DL}}$, for the conditions $\vec{c}^*$ created by some party P via $(\mathsf{create\text{-}1\text{-}of\text{-}n\text{-}cond}, (Y^*, y^*), \mathit{index}^*, n, \{P_i\})$ and $P \notin \mathcal{C}$, then the experiment aborts by outputting fail_2.

<u>Hybrid $\mathcal{H}_4$:</u> For the set of corrupted parties $\mathcal{C}$, if the adversary outputs $(\mathsf{open\text{-}cond}, (Y^*, y^*))$, such that $(Y^*, y^*) \in R_{\mathsf{DL}}$, for the condition Y^* created by some party P via $(\mathsf{create\text{-}merged\text{-}cond}, (Y_1^*, Y_2^*))$ and $P \notin \mathcal{C}$, then the experiment aborts by outputting fail_3.

<u>Simulator $\mathcal{S}$:</u> The simulator $\mathcal{S}$ simulates the honest parties as in the previous hybrid, except that its actions are dictated by the interaction with the ideal functionality $\mathcal{G}^{R, f_{\mathsf{merge}}}_{\mathsf{Cond}}$.

More precisely, the simulator proceeds as in the execution of hybrid $\mathcal{H}_4$ by simulating the view of the adversary appropriately as it receives messages from the ideal functionality $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$. If the simulated view deviates from the execution of the ideal functionality, then the simulation must have already aborted (as given in cases of abort in the above hybrids).

Next, we proceed to proving the indistinguishability of the neighboring hybrids for the environment $\mathcal{E}$.

Lemma 5. *For all* PPT *distinguishers* $\mathcal{E}$ *it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}.$$

Proof. The indistinguishability follows directly from the zero-knowledge property of the NIZK proof system Π_{NIZK}. More precisely, assume towards a contradiction that $\mathcal{E}$ can distinguish the two executions with a non-negligible probability. We give a reduction to the zero-knowledge property of Π_{NIZK}. The reduction sets the statement $x := (c[index] = g^{s[index]} \wedge (\forall j \in [n] \wedge j \neq index, c[j] = f_j \cdot g^{s[j]}))$, and sends it to the zero-knowledge challenger, which responds with a proof π_{index} that is either an honest proof or a simulated proof. The reduction then acts as the honest party P in its interaction with $\mathcal{E}$ during the creation of 1-out-of-n condition, computing everything as in hybrid $\mathcal{H}_0$, except that it uses the proof π_{index} it received from the zero-knowledge challenger. At the end of the execution, based on $\mathcal{E}$'s guess, it outputs a bit to the challenger (0 if $\mathcal{E}$ guesses hybrid $\mathcal{H}_0$, and 1 otherwise), which will be correct with non-negligible advantage. However, this violates the zero-knowledge property of Π_{NIZK}, and hence, the two executions must be indistinguishable. □

Lemma 6. *For all* PPT *distinguishers* $\mathcal{E}$ *it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}.$$

Proof. Let fail_1 be the event that triggers an abort in $\mathcal{H}_2$ but not in $\mathcal{H}_1$. In the following we are going to show that the probability that such an event happens can be bounded by a negligible function in the security parameter. Assume towards contradiction that $\Pr[\mathsf{fail}_1 \mid \mathcal{H}_1] \geq \frac{1}{\mathsf{poly}(\lambda)}$. To show that the probability of fail_1 happening in $\mathcal{H}_2$ cannot be inverse polynomial we reduce it to the hardness of the relation R_{DL} (as defined in Section 2.2.1). The reduction receives as input a group element Y, and samples an index $j \in [1, s]$, where $s \in \mathsf{poly}(\lambda)$ is a bound on the total number of sessions. The reduction sets the condition as $Y^* = Y$ in the j-th session. If the event fail_1 happens, then the reductions returns the corresponding y^*, otherwise it aborts.

The reduction is clearly efficient, and whenever j is guessed correctly it does not abort. Since fail_1 happens it means that $(Y^*, y^*) \in R_{\mathsf{DL}}$, for the condition Y^*, and $P \notin \mathcal{C}$. This implies that the reduction succeeded in breaking the hard relation R_{DL}. By assumption this happens with probability at least $\frac{1}{s \cdot \mathsf{poly}(\lambda)}$, which is a contradiction and proves that $\Pr[\mathsf{fail}_1 \mid \mathcal{H}_1] \leq \mathsf{negl}(\lambda)$. □

Lemma 7. *For all* PPT *distinguishers* $\mathcal{E}$ *it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}.$$

Proof. The proof of this lemma is analogous to that of Lemma 6, except that we need to account for index^*. More precisely, since $(\vec{c}^*[\mathit{index}^*], y^*) \in R_{\mathsf{DL}}$, we know that $\vec{c}^*[\mathit{index}^*] = g^{y^*}$. Hence, the reduction needs to additionally guess $\mathit{index}^* \in [n]$ before embedding the hard relation challenge. This incurs an additional $\frac{1}{n}$ loss, where $n \in \mathsf{poly}(\lambda)$. □

Lemma 8. *For all* PPT *distinguishers* $\mathcal{E}$ *it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}.$$

Proof. The proof of this lemma is analogous to that of Lemma 6. □

Lemma 9. *For all* PPT *distinguishers* $\mathcal{E}$ *it holds that*

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{G}^R_{\mathsf{Cond}},\mathcal{S},\mathcal{E}}.$$

Proof. The two experiments are identical and the change here is only syntactical. Hence, indistinguishability of the real world and ideal world follows. □

This concludes the proof of Theorem 3. □

4.2 Composable Adaptor Signatures

In contrast to the adaptor signature scheme whose security is characterized in terms of game-based security definitions (as given in Section 2.2.5), modeling adaptor signatures as an ideal functionality, denoted by $\mathcal{F}_{\mathsf{AdaptSig}}$, comes with the benefits of composable reasoning within the UC framework. In particular, it enables modular reasoning with respect to conditions, since our model of $\mathcal{F}_{\mathsf{AdaptSig}}$ relies on $\mathcal{G}_{\mathsf{Cond}}$ to handle conditions. This means that for signature adaptation $\mathcal{F}_{\mathsf{AdaptSig}}$ queries $\mathcal{G}_{\mathsf{Cond}}$ for determining whether the correct witness to open a condition was provided. We detail on this in Section 4.2.1.

In order to prove realization of $\mathcal{F}_{\mathsf{AdaptSig}}$ we create a wrapper protocol Π_{AdaptSig} around the two-party adaptor signature scheme with aggregatable public keys. We note that we can obtain two-party adaptor signature schemes generically by starting from any digital signature scheme obtained via an identification scheme[5], such as Schnorr [Sch91], Katz-Wang [KW03] or Guillou-Quisquater [GQ90] signatures, as shown by Erwig et

[5] We remark that our post-quantum adaptor signatures scheme from Chapter 3 is also obtained from an identification scheme, however, due to the limited algebraic structure present in group actions it does not fit the framework of Erwig et al. [EFH$^+$21].

al. [EFH$^+$21]. In Section 4.2.2 we show that Π_{AdaptSig} UC-realizes $\mathcal{F}_{\mathsf{AdaptSig}}$ in $\mathcal{G}_{\mathsf{Cond}}$-hybrid world. This realization is pictorially depicted in Figure 4.3.

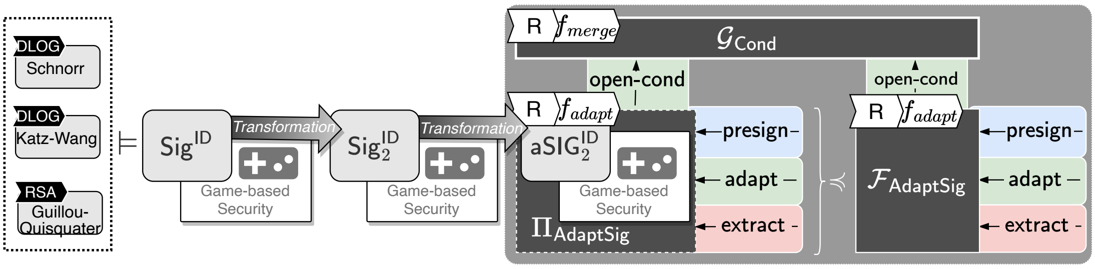


Figure 4.3: Realization of $\mathcal{F}_{\mathsf{AdaptSig}}$ in $\mathcal{G}_{\mathsf{Cond}}$-hybrid world.

4.2.1 Ideal Functionality $\mathcal{F}_{\mathsf{AdaptSig}}$

We first define the adaptor signatures ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$, which accounts for multiple keys per session and models two-party key generation (with public key aggregation) and signing. Then, we show that any two-party adaptor signature scheme with aggregatable public keys Ξ_{R,Σ_2} securely realizes the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$.

The functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ is depicted in Figure 4.4, and it extends the digital signature functionality given in [KRDO17]. First, our functionality accounts for multiple keys per session and models two-party key generation (with public key aggregation) and the two-party (pre-)signing protocol. Additionally, to account for adaptor signatures we parameterize the functionality with a hard relation R and a deterministic adaptation function f_{adapt}. The function f_{adapt} transforms a pre-signature $\hat{\sigma}$ and a witness y into a corresponding full signature σ.

The parametrization with f_{adapt} is required to stay fully modular with respect to $\mathcal{G}_{\mathsf{Cond}}$. By using $\mathcal{G}_{\mathsf{Cond}}$ in a modular fashion, we do not make any assumption about the creation of conditions, and in particular about which protocol parties know the witness for a condition. However, for parties knowing the witness y, the pre-signature $\hat{\sigma}$ and the corresponding adapted signature $\sigma := f_{\mathsf{adapt}}(\hat{\sigma}, y)$ are distinctly connected. For showing that a protocol Π_{AdaptSig} UC-realizes $\mathcal{F}_{\mathsf{AdaptSig}}$ without assuming any knowledge about which party knows y, we need to ensure that signatures created via the adaptation interface of $\mathcal{F}_{\mathsf{AdaptSig}}$ cannot be distinguished from those created through the adaptation algorithm (even for parties knowing y). If we let the simulator choose y at this point (as one would usually do in such cases), the simulator could not be guaranteed to know y, nor to complete the signature adequately (without leaking y or making an assumption on condition generation). Consequently, we need to let $\mathcal{F}_{\mathsf{AdaptSig}}$ compute the correct signature based on y, to which end we must fix f_{adapt}.

The functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ captures the expected correctness and security properties of a two-party adaptor signature as described in Section 2.2.6. More precisely, it captures completeness and consistency, along with (two-party) unforgeability and witness extractability. Pre-signature correctness and adaptability are captured with the help of the function f_{adapt} and global conditions functionality $\mathcal{G}_{\mathsf{Cond}}$.

Modeling Privacy of Adaptor Signatures. In a recent work, Dai et al. [DOY22] provided the first game-based privacy notion for adaptor signatures, named *unlinkability*. Intuitively, it ensures that adapted signatures cannot be distinguished from fresh full signatures. Moreover, Dai et al. [DOY22] showed that all the existing adaptor signatures achieve this property.

However, we note that our adaptor signature functionality from Figure 4.4 does not model any privacy property. For example, capturing the unlinkability property for two-party adaptor signatures is difficult because it inherently only holds against a third observer party, i.e., a party that is not involved in (pre-)signature generation and only sees the final signatures. Though, during the UC security proof we need to consider that the parties actually involved in the two-party (pre-)signing are adversarial, hence, such a third-party privacy notion is not provable.

More technically, in order to model such a privacy notion we need the simulator $\mathcal{S}$ to produce a valid full signature σ without having access to the corresponding witness y. One potential way to achieve this is inside the adaptation interface of $\mathcal{F}_{\mathsf{AdaptSig}}$ to make a call to the simulator $\mathcal{S}$, and let $\mathcal{S}$ return a fresh full signature σ' that is independent of the witness y. Then, we can argue that σ' is indistinguishable from a full signature σ, which is obtained by adapting a pre-signature $\hat{\sigma}$ with the witness y. However, since we are in the two-party adaptor signatures setting, it means that during the simulation we have to assume that one of the two parties involved in (pre-)signature generation is corrupted. Though, if one of the (pre-)signers is adversarial, then it is trivial for the adversary to distinguish different protocol runs, and hence, different signatures, since it is itself involved in the protocol execution and signature creation.

Nevertheless, we believe that a privacy notion, such as the unlinkability property of Dai et al. [DOY22], is capturable and provable within the UC framework if we consider regular (i.e., single party) adaptor signatures.

4.2.2 Protocol Π_{AdaptSig}

We depict in Figure 4.5 how to straightforwardly translate a two-party adaptor signature scheme with aggregatable public keys (obtained from an identification scheme) Ξ_{R,Σ_2} into a protocol $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$[6].

[6] Parameterizing the protocol with a deterministic adaptation function f_{adapt} is without loss of generality, since the generic transformation of Erwig et al. [EFH$^+$21, Figure 7] considers that the adaptation algorithm of two-party adaptor signature coincides with the function f_{adapt}.

Ideal Functionality $\mathcal{F}_{\mathsf{AdaptSig}}^{R,f_{\mathsf{adapt}}}$

The functionality is parameterized by a hard relation R and an adaptation function f_{adapt}. It maintains the list $\mathcal{K}$ that stores all generated keys, the list $\mathcal{Q}$ that stores tuples (m, σ, v, f) representing message, signature, key and a verification flag, and the list $\mathcal{P}$ that stores tuples $(m, \hat{\sigma}, \sigma, v, Y, y, f)$ representing message, pre-signature, signature, key, condition, witness, and pre-verification flag. All lists are indexed by a session identifier and are initially set to $\emptyset$.

Key Generation: Upon receiving $(\mathsf{keygen}, \mathsf{sid})$ from P_0 and P_1, verify that $\mathsf{sid} = (P_0, P_1, \mathsf{sid}')$ for some sid'. If not, ignore the request. Else, send $(\mathsf{keygen}, \mathsf{sid})$ to $\mathcal{S}$. Upon receiving $(\mathsf{verification\text{-}key}, \mathsf{sid}, v)$ from $\mathcal{S}$, add v into $\mathcal{K}[\mathsf{sid}]$ and send $(\mathsf{verification\text{-}key}, \mathsf{sid}, v)$ to P_0 and P_1.

Adaptation: Upon receiving $(\mathsf{adapt}, \mathsf{sid}, \hat{\sigma}, v, y)$ from some party P, check if there is an entry $\ell := (m, \hat{\sigma}, \bot, v, Y, \bot, 1) \in \mathcal{P}[\mathsf{sid}]$. If not, then ignore this request. Else, send $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y, y))$ to $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$. Upon receiving $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b)$, if $b = 0$, then abort. Else set $\sigma := f_{\mathsf{adapt}}(\hat{\sigma}, y)$ and update ℓ as $(m, \hat{\sigma}, \sigma, v, Y, y, 1)$ in $\mathcal{P}[\mathsf{sid}]$, add $(m, \sigma, v, 1)$ into $\mathcal{Q}[\mathsf{sid}]$, and send $(\mathsf{adapted\text{-}signature}, \mathsf{sid}, \sigma)$ to P. *(This guarantees pre-signature adaptability: any valid pre-signature $\hat{\sigma}$ can be adapted into a valid full signature σ using the witness y.)*

Extraction: Upon receiving $(\mathsf{extract}, \mathsf{sid}, \sigma, \hat{\sigma}, v)$ from some party P, check if there is an entry $(m, \hat{\sigma}, \sigma, v, Y, y, 1)$ in $\mathcal{P}[\mathsf{sid}]$. If not, then send $(\mathsf{witness}, \mathsf{sid}, \bot)$ to P, otherwise, send $(\mathsf{witness}, \mathsf{sid}, y)$ to P. *(This guarantees witness extractability: any valid signature/pre-signature pair $(\sigma, \hat{\sigma})$ can be used to extract the corresponding witness y.)*

(Pre-)Signature Generation: Upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \mathtt{type})$ from P_0 and P_1, verify that $\mathsf{sid} = (P_0, P_1, \mathsf{sid}')$ for some sid' and $v \in \mathcal{K}[\mathsf{sid}]$. If not, ignore the request. Else, send $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \mathtt{type})$ to $\mathcal{S}$. Upon receiving $(\mathsf{signature}, \mathsf{sid}, m, \sigma)$ from $\mathcal{S}$,

- if $\mathtt{type} = \textit{signature}$ and $(m, \sigma, v, 0) \notin \mathcal{Q}[\mathsf{sid}]$, then add $(m, \sigma, v, 1)$ into $\mathcal{Q}[\mathsf{sid}]$;
- if $\mathtt{type} = \textit{pre-signature}$ and $(m, \sigma, \bot, v, Y, \bot, 0) \notin \mathcal{P}[\mathsf{sid}]$, then add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, 1)$ into $\mathcal{P}[\mathsf{sid}]$.

If any of the above checks fail, then output an error and halt. Otherwise, output $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to P_0 and P_1.

(Pre-)Signature Verification: Upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \mathtt{type})$ from some party P, send $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \mathtt{type})$ to $\mathcal{S}$. Upon receiving $(\mathsf{verified}, \mathsf{sid}, m, \phi)$ from $\mathcal{S}$, do the following:

- If $v \in \mathcal{K}[\mathsf{sid}]$, and $(m, \sigma, v, 1) \in \mathcal{Q}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{signature}$) or $(m, \hat{\sigma} := \sigma, \cdot, v, Y, \cdot, 1) \in \mathcal{P}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{pre-signature}$), then set $f = 1$. *(This condition guarantees completeness: if the verification key v is registered before and σ is a legitimately generated (pre-)signature for m, then the verification succeeds.)*
- Else, if $v \in \mathcal{K}[\mathsf{sid}]$, the signers are not corrupted, and $(m, \sigma', v, 1) \notin \mathcal{Q}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{signature}$) or $(m, \sigma', \cdot, v, Y, \cdot, 1) \notin \mathcal{P}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{pre-signature}$) for any σ', then set $f = 0$ and add $(m, \sigma, v, 0)$ into $\mathcal{Q}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{signature}$) or add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, 0)$ into $\mathcal{P}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{pre-signature}$). *(This condition guarantees unforgeability: if v is one of the registered keys, the signers are not corrupted and never (pre-)signed m, then the verification fails.)*
- Else, if there exists $(m, \sigma, v, f') \in \mathcal{Q}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{signature}$) or $(m, \hat{\sigma} := \sigma, \cdot, v, Y, \cdot, f') \in \mathcal{P}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{pre-signature}$), then set $f = f'$. *(This guarantees consistency: all verification requests with identical parameters will result in the same answer.)*
- Else, set $f = \phi$, add (m, σ, v, ϕ) into $\mathcal{Q}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{signature}$) or add $(m, \hat{\sigma} := \sigma, \bot, v, Y, \bot, \phi)$ into $\mathcal{P}[\mathsf{sid}]$ (if $\mathtt{type} = \textit{pre-signature}$).

Output $(\mathsf{verified}, \mathsf{sid}, m, f)$ to P.

Figure 4.4: Ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}^{R,f_{\mathsf{adapt}}}$.

Note that the protocol $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ is only a generic wrapper protocol around the algorithmic interface of two-party adaptor signature scheme. Moreover, the protocol $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ is defined in $\mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$-hybrid world, where both Π_{AdaptSig} and $\mathcal{G}_{\mathsf{Cond}}$ consider the same hard relation R.

Protocol $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$

We consider parties P_b for $b \in \{0,1\}$ running signing and pre-signing in $\Pi^{f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$, and upon each request we verify that $\mathsf{sid} := (P_b, P_{1-b}, \mathsf{sid}')$ for some sid', and if not, then we ignore the request.

Key Generation: Party P_b upon receiving $(\mathsf{keygen}, \mathsf{sid})$ from $\mathcal{E}$:
- Generate keys $(\mathsf{sk}_b, \mathsf{pk}_b) \leftarrow \Sigma_2.\mathsf{KeyGen}(\mathsf{pp})$, for some public parameters pp.
- Store sk_b, and output $(\mathsf{verification\text{-}key}, \mathsf{sid}, v := (\mathsf{pk}_b, \mathsf{pk}_{1-b}))$.

Signing: Party P_b upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \textit{signature})$ from $\mathcal{E}$:
- Parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$.
- Execute the protocol $\sigma \leftarrow \Sigma_2.\Pi_{\mathsf{Sig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b}\rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, m)$.
- Output $(\mathsf{signature}, \mathsf{sid}, \sigma)$.

Pre-Signing: Party P_b upon receiving $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \textit{pre-signature})$ from $\mathcal{E}$:
- Parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and Y as Y.
- Execute the protocol $\hat{\sigma} \leftarrow \Xi_{R,\Sigma_2}.\Pi_{\mathsf{PreSig}\langle \mathsf{sk}_b, \mathsf{sk}_{1-b}\rangle}(\mathsf{pk}_b, \mathsf{pk}_{1-b}, m, Y)$.
- Output $(\mathsf{signature}, \mathsf{sid}, \hat{\sigma})$.

Verify: Party P upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \textit{signature})$ from $\mathcal{E}$:
- Parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$.
- Compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $f \leftarrow \Sigma_2.\mathsf{Ver}(\mathsf{apk}, m, \sigma)$.
- Output $(\mathsf{verified}, \mathsf{sid}, m, f)$.

Pre-Verify: Party P upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \hat{\sigma}, v, Y, \textit{pre-signature})$ from $\mathcal{E}$:
- Parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and Y as Y.
- Compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $f \leftarrow \Xi_{R,\Sigma_2}.\mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma})$.
- Output $(\mathsf{verified}, \mathsf{sid}, m, f)$.

Adaptation: Party P upon receiving $(\mathsf{adapt}, \mathsf{sid}, \hat{\sigma}, v, y, Y)$ from $\mathcal{E}$:
- Invoke $\mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$ on input $(\mathsf{open\text{-}cond}, \mathsf{sid}, Y, y)$.
- Receive $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b)$ from $\mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$.
- If $b = 0$, then abort. Else, compute $\sigma := f_{\mathsf{adapt}}(\hat{\sigma}, y)$.
- Output $(\mathsf{adapted\text{-}signature}, \mathsf{sid}, \sigma)$.

Extraction: Party P upon receiving $(\mathsf{extract}, \mathsf{sid}, \sigma, \hat{\sigma}, v)$ from $\mathcal{E}$:
- Parse v as $(\mathsf{pk}_b, \mathsf{pk}_{1-b})$.
- Compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_b, \mathsf{pk}_{1-b})$ and $y \leftarrow \Xi_{R,\Sigma_2}.\mathsf{Ext}(\mathsf{apk}, \sigma, \hat{\sigma}, Y)$.
- Output $(\mathsf{witness}, \mathsf{sid}, y)$.

Figure 4.5: Protocol $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ in $\mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$-hybrid world.

4.2.3 Security Proof

The security of our construction is established with the following theorem.

Theorem 4. *Let Ξ_{R,Σ_2} be a secure two-party adaptor signature scheme with aggregatable public keys (from an identification scheme) that is composed of a hard relation R and a secure two-party signature scheme Σ_2, then $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ UC-realizes $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.*

Proof. We give a proof by contradiction. Assume that $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ does not realize $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$, i.e., there exists an environment $\mathcal{E}$ that can differentiate whether it is interacting with $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ and $\mathcal{S}$ in the ideal world, or with $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ and $\mathcal{A}$ in the real world. We show that Ξ_{R,Σ_2} (used to obtain $\Pi^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$) violates the definition of a secure two-party adaptor signature scheme from Section 2.2.6. Since the environment $\mathcal{E}$ succeeds for any $\mathcal{S}$, it also succeeds for the following *generic* $\mathcal{S}$, denoted by $\mathcal{S}_{\mathsf{AdaptSig}}$, which runs a simulated copy of $\mathcal{A}$ and does the following (where $b \in \{0,1\}$ defines which of the two parties is corrupted):

Simulator $\mathcal{S}_{\mathsf{AdaptSig}}$

Any input from $\mathcal{E}$ is forwarded to $\mathcal{A}$, and any output from $\mathcal{A}$ is copied to $\mathcal{S}_{\mathsf{AdaptSig}}$'s output (to be read by $\mathcal{E}$). Moreover, when $\mathcal{A}$ corrupts some party P, then $\mathcal{S}_{\mathsf{AdaptSig}}$ corrupts P in the ideal world. If P is the signer, then $\mathcal{S}_{\mathsf{AdaptSig}}$ reveals the signing key sk (and the internal state of the signing algorithm, if such a state exists) as P's internal state.

Upon receiving a message $(\mathsf{keygen}, \mathsf{sid})$ from $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$:

1. If sid is not of the form $(P_{1-b}, P_b, \mathsf{sid}')$, then ignore the request.
2. Else, run $(\mathsf{sk}_{1-b}, \mathsf{pk}_{1-b}) \leftarrow \Sigma_2.\mathsf{KeyGen}(\mathsf{pp})$, set $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, record the tuple $(\mathsf{sid}, \mathsf{sk}_{1-b}, v)$, and send $(\mathsf{verification\text{-}key}, \mathsf{sid}, v)$ to $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.

Upon receiving a message $(\mathsf{sign}, \mathsf{sid}, m, v, Y, \mathsf{type})$ from $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$:

1. If sid is not of the form $(P_{1-b}, P_b, \mathsf{sid}')$ and no tuple of the form $(\mathsf{sid}, \mathsf{sk}_{1-b}, v)$, has been previously recorded, then ignore the request.
2. If $\mathsf{type} = \mathsf{signature}$, then parse $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, simulate a run of the protocol $\sigma \leftarrow \Sigma_2.\Pi^{\mathcal{A}}_{\mathsf{Sig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_{1-b}, \mathsf{pk}_b, m)$ (by simulating the honest party P_{1-b}), and send $(\mathsf{signature}, \mathsf{sid}, m, \sigma)$ to $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.
3. If $\mathsf{type} = \mathsf{pre\text{-}signature}$, then parse $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, simulate a run of the protocol $\hat{\sigma} \leftarrow \Xi_{R,\Sigma_2}.\Pi^{\mathcal{A}}_{\mathsf{PreSig}\langle \mathsf{sk}_{1-b}, \cdot \rangle}(\mathsf{pk}_{1-b}, \mathsf{pk}_b, m, Y)$ (by simulating the honest party P_{1-b}), and send $(\mathsf{signature}, \mathsf{sid}, m, \hat{\sigma})$ to $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.

Upon receiving $(\mathsf{verify}, \mathsf{sid}, m, \sigma, v, Y, \mathsf{type})$ from $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$:

1. If $\mathsf{type} = \mathsf{signature}$, then parse $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, set $\phi := \Sigma_2.\mathsf{Ver}(\mathsf{apk}, m, \sigma)$, and send $(\mathsf{verified}, \mathsf{sid}, m, \phi)$ to $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.
2. If $\mathsf{type} = \mathsf{pre\text{-}signature}$, parse $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, compute $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, set $\phi := \Xi_{R,\Sigma_2}.\mathsf{PreVer}(\mathsf{apk}, m, Y, \hat{\sigma})$, and send $(\mathsf{verified}, \mathsf{sid}, m, \phi)$ to $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.

Next, we assume that Ξ_{R,Σ_2} is both complete and consistent, and construct an attacker $\mathcal{B}$ that breaks the unforgeability. $\mathcal{B}$ runs a simulated copy of $\mathcal{E}$ and simulates the interactions with $\mathcal{S}_{\mathsf{AdaptSig}}$ and $\mathcal{F}_{\mathsf{AdaptSig}}$. Analogous to $\mathcal{S}_{\mathsf{AdaptSig}}$, $\mathcal{B}$ also runs a simulated copy of $\mathcal{A}$. However, in the first activation, instead of running $\Sigma_2.\mathsf{KeyGen}$ to generate the key, $\mathcal{B}$ gives to $\mathcal{A}$ the verification key pk_{1-b} that it has received as an input from its own challenger (where $b \in \{0,1\}$ defines which of the two parties is corrupted). Whenever $\mathcal{B}$ needs to provide (pre-)signature on a message m, $\mathcal{B}$ asks its oracles to (pre-)sign m and obtains (pre-)signature σ. Moreover, $\mathcal{B}$ and $\mathcal{A}$ jointly generate pre-signature $\hat{\sigma}^*$ on the challenge message m^* (provided by $\mathcal{A}$), where $\mathcal{B}$ just relays the protocol messages of its own challenger when computing the joint pre-signature on the same challenge message m^*. Finally, whenever the simulated $\mathcal{E}$ activates some party with input $(\mathsf{verify}, \mathsf{sid}, m^*, \sigma^*, v, Y, \mathsf{type})$, where $\mathsf{type} = \mathsf{signature}$, $\mathcal{B}$ checks whether (m^*, σ^*) constitutes a valid forgery (i.e., m^* is the challenge message and it has never been signed before and $\Sigma_2.\mathsf{Ver}(\mathsf{apk}, m^*, \sigma^*) = 1$, for $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, where $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$ and pk_b is the verification key of $\mathcal{A}$). If (m^*, σ^*) is a valid forgery, then $\mathcal{B}$ outputs σ^* as its own forgery and halts. Otherwise, it continues the simulation. If $\mathcal{A}$ asks to corrupt the honest signer P_{1-b}, then $\mathcal{B}$ halts with a failure.

We analyze the success probability of $\mathcal{B}$. Let win denote the event that in a run of Ξ_{R,Σ_2} with $\mathcal{A}$ and $\mathcal{E}$ with $\mathsf{sid} = (P_{1-b}, P_b, \mathsf{sid}')$, the signers generate the verification keys pk_{1-b} and pk_b, such that $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, some party is activated with verification request $(\mathsf{verify}, \mathsf{sid}, m^*, \sigma^*, v, Y, \mathsf{type})$, where $\mathsf{type} = \mathsf{signature}$ and $\Sigma_2.\mathsf{Ver}(v, m^*, \sigma^*) = 1$, party P_{1-b} is uncorrupted and signers never signed m^*. Since Ξ_{R,Σ_2} is complete and consistent, we have that as long as the event win does not occur $\mathcal{E}$'s view of an interaction with $\mathcal{A}$ and Ξ_{R,Σ_2} in the real world is statistically close to its view of an interaction with $\mathcal{S}_{\mathsf{AdaptSig}}$ and $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ in the ideal world. However, we are given that $\mathcal{E}$ distinguishes with non-negligible probability between the ideal and real world, thus, we are guaranteed that when $\mathcal{E}$ interacts with $\mathcal{A}$ and Ξ_{R,Σ_2}, event win occurs with non-negligible probability.

Finally, observe that from the point of view of $\mathcal{A}$ and $\mathcal{E}$, the interaction with the forger $\mathcal{B}$ looks the same as an interaction in the real world with Ξ_{R,Σ_2}. Hence, we are guaranteed that event win occurs with non-negligible probability. Furthermore, event win can only occur when P_{1-b} is not corrupted. This means whenever event win occurs, $\mathcal{B}$ outputs a successful forgery, which contradicts the unforgeability definition of Ξ_{R,Σ_2}.

We can construct a similar adversary $\mathcal{B}'$ against the witness extractability property of Ξ_{R,Σ_2}. $\mathcal{B}'$ works exactly as $\mathcal{B}$ above, with the difference that the joint pre-signature $\hat{\sigma}^*$ is computed over both the challenge message m^* and challenge statement Y^* provided by the adversary $\mathcal{A}$. Moreover, the winning condition is adjusted such that m^* is the challenge message and it has never been signed before and $\Sigma_2.\mathsf{Ver}(\mathsf{apk}, m^*, \sigma^*) = 1$, for $\mathsf{apk} := \Sigma_2.\mathsf{KAgg}(\mathsf{pk}_{1-b}, \mathsf{pk}_b)$, where $v := (\mathsf{pk}_{1-b}, \mathsf{pk}_b)$ and pk_b is the verification key of $\mathcal{A}$, and $(Y^*, y') \notin R$, for $y' := \Xi_{R,\Sigma_2}.\mathsf{Ext}(v, \sigma^*, \hat{\sigma}^*, Y^*)$.

Lastly, we observe that pre-signature adaptability is captured within the adaptation interface of the ideal functionality $\mathcal{F}_{\mathsf{AdaptSig}}$, which in turn makes use of the global ideal functionality $\mathcal{G}_{\mathsf{Cond}}$ and its parameterized deterministic adaptation function f_{adapt}. More precisely, a valid pre-signature $\hat{\sigma}$ w.r.t. a condition Y can be adapted into a valid signature, i.e., $\sigma := f_{\mathsf{adapt}}(\hat{\sigma}, y)$, using the witness y that satisfies $(Y, y) \in R$.

This concludes the proof of Theorem 4. □

CHAPTER 5

Security Framework for Protocols Based on Adaptor Signatures

In this chapter we present our framework, named LedgerLocks, for proving security of blockchain protocol based on adaptor signatures. First, in Section 5.1 we discuss the previous approaches for analyzing the security of blockchain protocols based on adaptor signatures and motivate the need for our framework. Then, in Section 5.2 we formally describe our framework and in Section 5.3 we detail how one can use our framework to model and prove security of adaptor signature-based blockchain protocols.

5.1 Previous Approaches for Blockchain Protocol Analysis

In this section, we overview the existing approaches to analyze the security of adaptor signature-based (AS-based) blockchain protocols with an emphasis on the ledger modeling. For this, we first give background on the workings of realistic ledgers and then illustrate the impact of the ledger model on the security analysis using the examples of an atomic swap and a multi-hop off-chain payment protocol. Finally, we discuss the ledger models used in literature and their limitations.

Blockchain Workings. As described in Section 2.4, in cryptocurrencies built upon a tamper-resistant distributed ledger (i.e., blockchain), network participants conduct transactions by broadcasting them to the network. Specific network nodes, so-called *miners*, group valid transactions into blocks and append them to the blockchain. To ensure fairness, the miner selection process is randomized based on a resource in the possession of miners (usually their computational power or financial stakes). If the majority of the resource is owned by honest miners, then it is guaranteed that the system will progress safely. More precisely, the system will reach consensus on a stable prefix of the blockchain and valid transactions are guaranteed to be eventually included in such a stable prefix.

The resulting transaction execution model comes with several peculiarities. For example, transactions submitted by honest users are not necessarily guaranteed to be included in the blockchain but could still be outrun by (adversarial) transactions that invalidate them, e.g., by consuming the same assets. Additionally, transactions are already public before their inclusion in the blockchain, possibly leaking sensitive information to a (miner-controlling) attacker.

Atomic Swaps. An *atomic swap* (shown in Figure 5.1) involves two ledgers $\mathbb{A}$ (blue), $\mathbb{B}$ (orange) and two users Alice (A) and Bob (B), holding assets in $\mathbb{A}$ and $\mathbb{B}$, respectively. A correct atomic swap protocol ensures that Alice receives Bob's assets on $\mathbb{B}$ and Bob receives Alice's assets on $\mathbb{A}$ if both parties are honest. An atomic swap protocol is considered secure if an honest party always either (i) receives the other party's assets; or (ii) keeps their own assets.

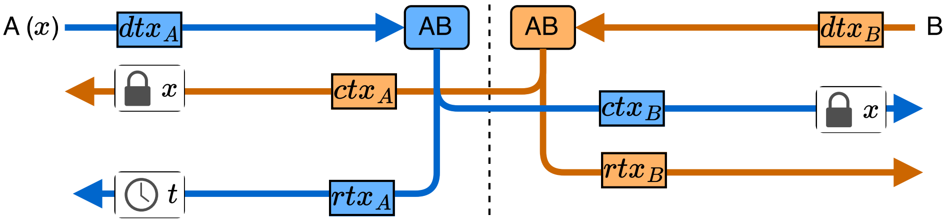


Figure 5.1: Transactions in an atomic swap protocol.

To set up such an atomic swap, Alice locally creates a cryptographic secret x. Moreover, Alice and Bob deposit their assets into a shared account $\boxed{\text{AB}}$ in the respective chain (through deposit transaction $\boxed{dtx_A}$ and $\boxed{dtx_B}$). The assets in a shared account are set up such that they can be released by the intended receiver showing the secret value x or refunded to the original owner. This functionality is achieved by Alice and Bob jointly creating AS-locked transactions $\boxed{ctx_A}$ and $\boxed{ctx_B}$, which can only be submitted upon the knowledge of secret x and whose publication will release x to the other party. Furthermore, they create the refund transactions $\boxed{rtx_A}$ and $\boxed{rtx_B}$ that allow Alice and Bob to retrieve back their assets in case the other party stops collaborating.

After a successful setup, Alice, who knows the secret x, can claim Bob's assets in $\mathbb{B}$ (using $\boxed{ctx_A}$). Then, Bob can read x from $\mathbb{B}$ and use it to claim the assets in the shared account on $\mathbb{A}$ (using $\boxed{ctx_B}$). Alternatively (e.g., if the other user fails to cooperate), Alice and Bob can refund their assets by publishing $\boxed{rtx_A}$ or $\boxed{rtx_B}$, respectively. Alice's refund transaction $\boxed{rtx_A}$ is equipped with a *timelock* that ensures that it can only be published after time t. This restriction prevents Alice from simultaneously publishing $\boxed{rtx_A}$ and $\boxed{ctx_A}$ to retrieve the assets on both chains. Instead, if Alice has not claimed Bob's assets (through $\boxed{ctx_A}$) until time right before t, Bob can publish $\boxed{rtx_B}$ to be refunded before $\boxed{rtx_A}$ becomes valid at time t.

Although the idea behind the atomic swap protocol seems simple, it is only secure when assuming a highly simplified ledger model. More precisely, its security relies on the assumption that $\boxed{rtx_B}$ will be included immediately after Bob sent it to the network. In practice, the ledger only guarantees, that $\boxed{rtx_B}$ will be included in the blockchain within a time delay Δ. During this time, other transactions (even if submitted later) may be included in the blockchain, invalidate $\boxed{rtx_B}$ and, thus, prevent its inclusion. Specifically, a malicious Alice could send $\boxed{ctx_B}$ right after observing $\boxed{rtx_B}$ on the network. As a consequence, $\boxed{ctx_B}$ could be included in the blockchain first, canceling $\boxed{rtx_B}$.

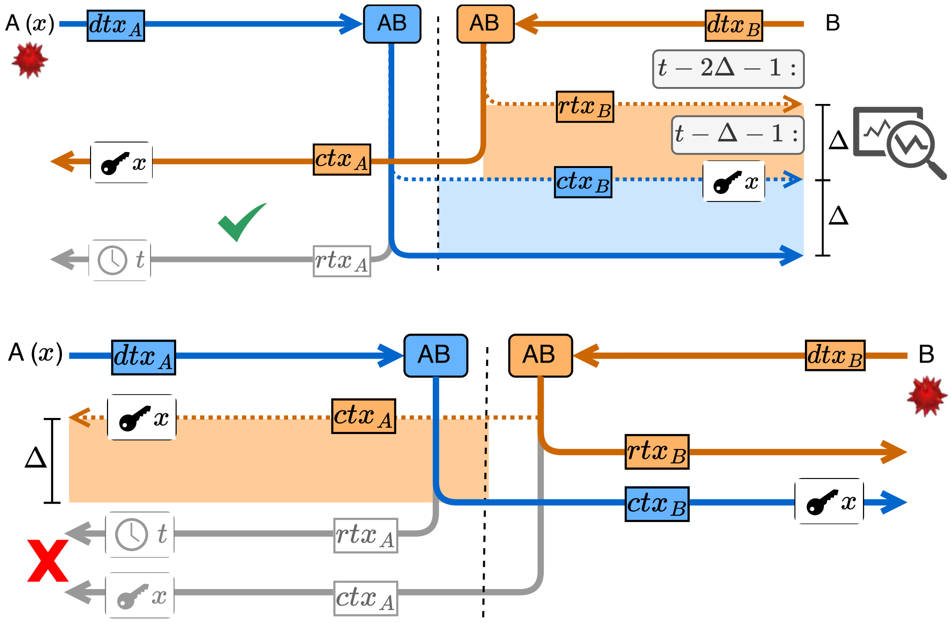


Figure 5.2: Atomic swap in a ledger with delayed inclusion (top). Attack in an atomic swap in a realistic ledger (bottom).

To secure the atomic swap protocol in the presence of such ledgers, Bob's behavior in the protocol needs to be adapted as illustrated in Figure 5.2 (top). More precisely, right before time $t - 2\Delta$, Bob needs to initiate the refund of their assets by publishing $\boxed{rtx_B}$ (denoted by a dotted arrow). Like this, Bob knows right before time $t - \Delta$ whether the refund was successful or whether Alice managed to outrun Bob with $\boxed{ctx_A}$. In the latter case, Bob learns x and publishes $\boxed{ctx_B}$, which is guaranteed to be included before t, the time starting from which Alice could publish $\boxed{rtx_A}$.

Interestingly, even the adapted protocol is insecure when considering another subtlety of realistic ledger workings. A transaction, once submitted to the network, becomes public to the miners even before being included in the blockchain. Considering that and despite her advantage of exclusively knowing secret x, Alice is subject to an attack (Figure 5.2, bottom). When Alice claims Bob's assets (by publishing $\boxed{ctx_A}$), a malicious Bob can learn x and still outrun $\boxed{ctx_A}$ with $\boxed{rtx_B}$. Then, Bob could claim Alice's assets (publishing $\boxed{ctx_B}$) before Alice could refund her assets using $\boxed{rtx_A}$ at time t.

Multi-Hop Payments. The aforedescribed issues do not only apply to atomic swaps but extend to a wide range of (AS-based) blockchain protocols. Another example is off-chain payments in payment channel networks such as described in [MMS+19]. Payment channel networks allow parties that do not share a payment channel to still securely exchange off-chain payments as long as they are connected via a path in a *payment channel network* (for background on payment channels refer to Section 2.4).

For preparing a payment, the parties P_i on a payment path between sender S $(= P_0)$ and receiver R $(= P_n)$ lock channel funds for the payment such that the payment later can be enforced atomically. To this end, the parties P_i $(0 \leq i < n)$ prepare *conditional off-chain payments* based on some condition c_i to their successors P_{i+1} on the payment path. These conditional off-chain payments are realized through AS-locked transactions $\boxed{\texttt{ctx}_i}$ on the channel funds that can be published once the channel is closed. The conditions c_i are set up such that if party P_{i+1} claims $\boxed{\texttt{ctx}_i}$, this will reveal a secret x_i to P_i allowing them to satisfy c_{i-1} and claim $\boxed{\texttt{ctx}_{i-1}}$ in turn. Once all conditional payments are set up, S initiates the payment by revealing a secret s_R to R that allows them to open c_{n-1}. Next, the payment gets propagated through the payment path, where collaborative parties P_i and P_{i+1} can update their payment channels off-chain after revealing the secret that would enable $\boxed{\texttt{ctx}_i}$. If P_i is not collaborating, P_{i+1} can close the channel and use $\boxed{\texttt{ctx}_i}$ to enforce the conditional payment based on the last channel state.

To ensure that a malicious sender cannot indefinitely lock the funds of intermediaries on the path, the parties, in addition to $\boxed{\texttt{ctx}_i}$ prepare a refund transaction $\boxed{\texttt{rtx}_i}$ that allows P_i to retrieve back their funds after time t_i. As for the atomic swap protocol, such a refund option introduces additional challenges in the design of a secure protocol. More precisely, if party P_{i+1} is not responding, honest P_i needs to close the channel and invoke the refund using $\boxed{\texttt{rtx}_i}$. However, even after successful channel closure while waiting for $\boxed{\texttt{rtx}_i}$ to be included in the blockchain, P_{i+1} may still decide to publish $\boxed{\texttt{ctx}_i}$ instead. In this case P_i needs to observe $\boxed{\texttt{ctx}_i}$ on the blockchain, learn x_i and use it to continue the payment (either off-chain or on-chain). For this, it needs to be ensured that $\boxed{\texttt{ctx}_{i-1}}$ is still valid at this point and so that $\boxed{\texttt{rtx}_{i-1}}$ has not been published yet.

This illustrates how the protocol design is closely intertwined with the precise guarantees that the underlying ledger provides:

1. The protocol transactions need to have timelocks that respect the ledger inclusion times. In particular, timelock t_i of $\boxed{\texttt{rtx}_i}$ needs to be adjusted such that $t_i < t_{i-1} + 2\Delta + \Delta_{close}$ (for t_{i-1} being the timelock of $\boxed{\texttt{rtx}_{i-1}}$ and Δ_{close} being the channel closing time) to ensure that u_i after closing their outgoing channel and publishing $\boxed{\texttt{rtx}_i}$ at t_i, when learning (latest) at $t_i + \Delta$ whether $\boxed{\texttt{rtx}_i}$ or $\boxed{\texttt{ctx}_i}$ got included in the blockchain there is still enough time to close their ingoing channel and publish $\boxed{\texttt{ctx}_{i-1}}$ on the blockchain (which may take up to $\Delta_{close} + \Delta$).

2. The honest participant strategies need to respect the timing constraints. For example, it is crucial that honest participants frequently poll the blockchain for the inclusion of payment channel closing transactions and to react on the publication of a claim transaction $\boxed{\texttt{ctx}}$ on-chain in well-defined time windows to obtain the desired correctness guarantees.

If the ledger model is not accounting for the exact ledger behavior concerning the attacker's delay and learning capabilities, wrong protocols can easily be proven secure. For example, consider a version of the multi-hop payment protocol where receiver R accepts S's payment too late. After setting up the payment, if R receives s_R only after $t_n + \Delta_{close}$, then R is not guaranteed anymore to receive the payment. If P_{n-1} does not collaborate in updating the channel, R needs to close the channel with P_{n-1} (taking up to Δ_{close}) and then publish $\boxed{\mathtt{ctx}_n}$ at time $t < t_n$. However, at t_n a malicious P_{n-1} already submitted $\boxed{\mathtt{rtx}_n}$ to outrun $\boxed{\mathtt{ctx}_n}$ resulting in P_{n-1} being refunded while still learning s_R. With the knowledge of s_R, P_{n-1} completes the payment and receives the funds meant for R. Similar to the atomic swap example, such an attack could not be detected in the presence of a ledger model with instant transaction inclusion or without modeling that the attacker may learn transaction details (such as x_i) before the transaction's inclusion in the blockchain.

State-of-the-Art Ledger Models. As highlighted by the examples in the last paragraph, there are several realistic ledger features whose modeling comes with immediate security implications. Foremost, this is a realistic attacker model that accounts for both the *attacker knowledge* (e.g., the knowledge of transactions after they got submitted but before they got included in the blockchain) and the *attacker capabilities* (e.g., to influence the order and time of transaction inclusion). Related to the attacker capabilities, the concrete *inclusion time guarantees* for honest users are crucial for secure protocol design (e.g., for the correct adjustment of timeouts).

The importance of a realistic ledger model for the security analysis of blockchain protocols is also emphasized by Kiayias and Litos [KL20], who propose a formal security analysis of Bitcoin's Lightning Network in the presence of the ledger model $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17]. The authors showcase that for precisely specifying the Lightning Network protocol, it is inevitable to rely on the exact timing guarantees obtained from the ledger in [BMTZ17]. Moreover, Kiayias and Litos [KL20] also prove that the simplified models that are used in the security analysis of [DFH18, DEF18, DEFM19, EMM19, TMM22] not only fail to reflect the guarantees of realistic ledgers but that it is impossible to design a ledger that could provide such guarantees. This problem was further studied by Badertscher, Hesse and Zikas [BHZ21], who showed that generic security-preserving replacement of ledger functionalities by their protocol implementation can only work under (often unrealistic) strong conditions on the underlying ledger.

Ledger Features	$\mathcal{L}_{inst}$	$\mathcal{L}_{\Delta}$	$\mathcal{G}_{\mathsf{Ledger}}$ / $\mathcal{G}_{\mathsf{LedgerLocks}}$
Attacker knowledge	✗	✗*	✓
Attacker capabilities	✗	✗*	✓
Inclusion time guarantees	✗	✓	✓
Realizability	✗	✗†	✓

Table 5.1: Overview of features of ledger models used for the analysis of blockchain protocols. ✗* denotes that the corresponding ledger feature is underspecified, ✗† indicates that the realizability of $\mathcal{L}_{\Delta}$ is unknown.

However, as summarized in Table 5.1, the state-of-the-art still analyses blockchain protocols in the presence of simplified ledger models, which disregard security-relevant ledger features. These works consider either

1. (provable unrealizable) ledgers $\mathcal{L}_{inst}$ with immediate inclusion guarantees [DFH18, DEF18, DEFM19, EMM19, TMM22]; or
2. ledgers $\mathcal{L}_{\Delta}$ that let the attacker delay the inclusion of a transaction up to some delay parameter Δ [TMSS20, AEE^{+}21, AME^{+}21, TMM21a, AMKM21b, GMM^{+}22, ATM^{+}22, TM21, QPM^{+}23].

For example, in [AEE^{+}21], it is stated that upon a message being posted by the user, the ledger should "wait until round $\tau_1 \leq \tau_0 + \Delta$ (the exact value of τ_1 is determined by the adversary)". This description leaves open at which point in time and based on which information the adversary determines the inclusion time τ_1. Though, as shown for the example of atomic swap and multi-hop payments, leaving these aspects underspecified may result in the security analysis missing relevant attacks.

Hence, the aim of our framework is to simplify the design of AS-based blockchain protocols in the presence of a realistic ledger. Towards this end, we propose the LedgerLocks framework, which extends the realistic ledger model $\mathcal{G}_{\mathsf{Ledger}}$ from [BMTZ17] to include an abstraction for the cryptographic operations required to synchronize transactions in AS-based protocols. In this way, the security analysis of these protocols can focus on the ledger-specific aspects instead of cryptographic arguments.

5.2 Lock-Enabling Ledger

In Section 5.2.1, we define $\mathcal{G}_{\mathsf{LedgerLocks}}$, an ideal functionality for a distributed ledger with AS-locked transactions. In the design of $\mathcal{G}_{\mathsf{LedgerLocks}}$, we follow the technique from Badertscher et al. [BMTZ17], where the authors provide $\mathcal{G}_{\mathsf{Ledger}}$, an ideal functionality modeling the subtleties of real-world blockchain consensus, in particular, realistic guarantees about the inclusion of transactions into the ledger. Moreover, they give Π_{Ledger}, a description of the Bitcoin backbone protocol and prove that it UC-realizes $\mathcal{G}_{\mathsf{Ledger}}$.

We note that $\mathcal{G}_{\mathsf{Ledger}}$ and Π_{Ledger} are generic in that they do not fix the concrete transaction format or ledger logic. Instead, both of them are parametrized with a predicate $\mathsf{isValidTx}$, which based on the internal ledger state determines whether a transaction is valid or not.

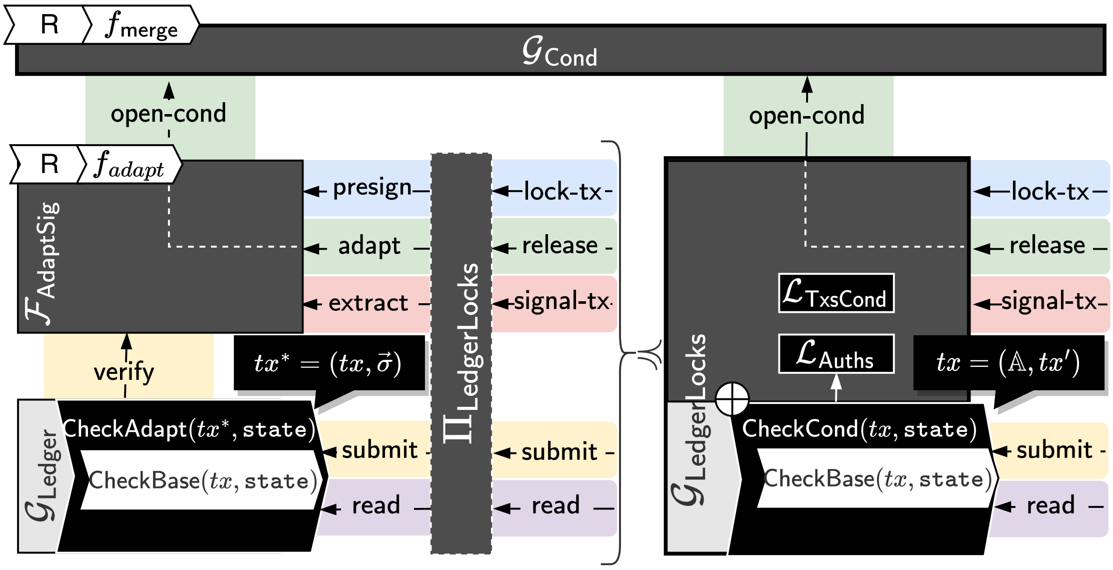
In this manner, the UC-realization proof holds for any instantiation of this predicate. Furthermore, one can leverage the results in [BMTZ17] by extending $\mathcal{G}_{\mathsf{Ledger}}$ in two ways:

1. instantiating $\mathsf{isValidTx}$ predicate to account for the specific transaction formats and ledger logics; and
2. extending $\mathcal{G}_{\mathsf{Ledger}}$ with additional interfaces to account for further ledger features.

Our ideal functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$ follows this blueprint to model multi-party account-based transaction authorization. In more detail, $\mathcal{G}_{\mathsf{LedgerLocks}}$ allows multiple parties to create a joint account. Transactions are associated with the set of all accounts, which need to provide authorization for transaction publication on the ledger. In addition to full authorization, accounts can lock a transaction on a condition, in which case any account owner can complete the authorization by providing an appropriate witness. If such an AS-locked transaction is published on the ledger, honest account owners learn the corresponding witness, while a malicious owner learns the witness already upon transaction submission.

We build $\mathcal{G}_{\mathsf{LedgerLocks}}$ from $\mathcal{G}_{\mathsf{Ledger}}$ by (i) requiring the transaction format to include the list of accounts to authorize the transaction; (ii) adding additional state and interfaces for the new operations; and (iii) instantiating the validity predicate to check for correct transaction authorization. The operation for releasing a AS-locked transaction thereby makes use of $\mathcal{G}_{\mathsf{Cond}}$ to determine whether a provided witness satisfies the condition of the corresponding transaction. To stay general, we do not fully fix the transaction format and the $\mathsf{isValidTx}$ predicate but introduce another predicate $\mathsf{CheckBase}$, which performs additional transaction validity checks. In this way, we can modularly add additional functionality to $\mathcal{G}_{\mathsf{LedgerLocks}}$, e.g., support for timelocks as we show in Section 6.1.

Finally, we show how to realize $\mathcal{G}_{\mathsf{LedgerLocks}}$ with a protocol $\Pi_{\mathsf{LedgerLocks}}$, which uses $\mathcal{G}_{\mathsf{Ledger}}$ and $\mathcal{F}_{\mathsf{AdaptSig}}$. Thanks to our modeling of $\mathcal{G}_{\mathsf{Cond}}$ as global ideal functionality, the whole construction and proof are independent of the concrete realization of conditions. We depict this realization pictorially in Figure 5.3, and refer to Sections 5.2.1 and 5.2.2 for more details.



R
f_{merge}
$\mathcal{G}_{\mathsf{Cond}}$
open-cond
open-cond
R
f_{adapt}
$\mathcal{F}_{\mathsf{AdaptSig}}$
presign
adapt
extract
verify
$tx^* = (tx, \vec{\sigma})$
$\mathsf{CheckAdapt}(tx^*, \mathtt{state})$
$\mathsf{CheckBase}(tx, \mathtt{state})$
$\mathcal{G}_{\mathsf{Ledger}}$
submit
read
$\Pi_{\mathsf{LedgerLocks}}$
lock-tx
release
signal-tx
submit
read
$\mathcal{L}_{\mathsf{TxsCond}}$
$\mathcal{L}_{\mathsf{Auths}}$
$tx = (\mathbb{A}, tx')$
$\mathsf{CheckCond}(tx, \mathtt{state})$
$\mathsf{CheckBase}(tx, \mathtt{state})$
$\mathcal{G}_{\mathsf{LedgerLocks}}$
lock-tx
release
signal-tx
submit
read

Figure 5.3: Realization of $\mathcal{G}_{\mathsf{LedgerLocks}}$ in $(\mathcal{F}_{\mathsf{AdaptSig}}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid world.

5.2.1 Ideal Functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$

We formally describe functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$ in Figure 5.4, which extends the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17] (shown in Section 2.4) with accounts and conditional payments. In Figure 5.3 we depict pictorially how $\mathcal{G}_{\mathsf{LedgerLocks}}$ is constructed from the base ledger $\mathcal{G}_{\mathsf{Ledger}}$ [BMTZ17], and in the following we explain the details of the functionality.

The functionality maintains a list $\mathcal{L}_{\mathsf{AccId}}$, that contains the generated accounts. Additionally, it uses $\mathcal{L}_{\mathsf{Auths}}$ that contains authorized transactions, and $\mathcal{L}_{\mathsf{TxsCond}}$ that contains conditional transactions. $\mathcal{G}_{\mathsf{LedgerLocks}}$ builds upon $\mathcal{G}_{\mathsf{Ledger}}$ by adding interfaces, introducing an additional state, and refining the transaction validation check (by instantiating the predicate $\mathsf{isValidTx}$). $\mathcal{G}_{\mathsf{LedgerLocks}}$ keeps three lists to model account management ($\mathcal{L}_{\mathsf{AccId}}$), authorization ($\mathcal{L}_{\mathsf{Auths}}$) and locking of transactions ($\mathcal{L}_{\mathsf{TxsCond}}$). The new validity predicate $\mathsf{CheckCond}$ operates on transactions of the form $(\mathbb{A}, \mathtt{tx}')$ where $\mathbb{A}$ denotes the set of accounts controlling the transaction $\mathtt{tx}'$. For checking the validity of a transaction, it checks $\mathcal{L}_{\mathsf{Auths}}$ for authorization of all accounts in $\mathbb{A}$ and invokes $\mathsf{CheckBase}$ for further validity checks. In addition to the interfaces for submitting and reading transactions provided by $\mathcal{G}_{\mathsf{Ledger}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$ provides five interfaces. The account generation interface allows multiple parties to jointly generate an account (added to $\mathcal{L}_{\mathsf{AccId}}$). Although the ideal functionality models multi-party account generation, we note that in our protocol (described in Section 5.2.2), we only consider two-party account generation as this is sufficient for our envisioned applications. Moreover, a transaction can have several accounts associated to it, contributing to the generality of the ideal functionality definition. In particular, this allows for modeling UTXO-style cryptocurrencies, where a transaction refers to multiple inputs, which may be controlled by different accounts.

The transaction locking interface allows the parties owning an account to jointly create an authorization for a transaction, which is locked under a specified condition and can only be released using the opening information for this condition. This authorization is recorded in $\mathcal{L}_{\mathsf{TxsCond}}$. The transaction release interface allows a party controlling the respective account and knowing the opening information of the condition to submit a locked transaction to the ledger, moving the transaction from $\mathcal{L}_{\mathsf{TxsCond}}$ to $\mathcal{L}_{\mathsf{Auths}}$. The witness signaling interface allows the account parties to extract the condition witness from the published AS-locked transaction.

One subtlety in our model here is that witness signaling is only enabled when the previously released transaction is added to the ledger. Only then we can guarantee that any party (involved in its creation) would have seen both the AS-locked transaction and the released transaction. We model this by checking whether the released transaction is in the ledger's view of the party invoking the witness signaling interface, which can be accessed by the ledger state variable, as defined in $\mathcal{G}_{\mathsf{Ledger}}$. While an honest user is only guaranteed to learn the witness upon inclusion of the transaction in the ledger, a malicious user may learn the witness already upon the transaction's submission to the ledger. As motivated in Section 5.1, this situation may occur if the attacker controls both the user participating in the creation of the AS-locked transaction and a miner. We reflect this subtlety in the release interface as follows: if any of the transaction's account owners is corrupted, then the witness is immediately sent to the adversary. As described in Section 5.1, modeling this behavior is crucial for an accurate security analysis of blockchain protocols.

5.2.2 Protocol $\Pi^{R}_{\mathsf{LedgerLocks}}$

Our lock-enabling ledger protocol $\Pi^{R}_{\mathsf{LedgerLocks}}$ is defined in the $(\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid model and given in Figure 5.5.

During account generation, parties obtain verification keys by making calls to the key generation interface of $\mathcal{F}_{\mathsf{AdaptSig}}$. Analogously, authorization of transactions and locking of transactions happen with a call to the signing interface of $\mathcal{F}_{\mathsf{AdaptSig}}$, where in the latter case, only a pre-signature $\hat{\sigma}$ that is conditioned on Y is computed, whereas in the former a full signature σ over the transaction is computed. Releasing of transactions happens by calling the adaptation interface of $\mathcal{F}_{\mathsf{AdaptSig}}$ with the witness (i.e., opening) y of the corresponding condition (i.e., statement) Y used during the locking procedure. Lastly, witness signaling calls the extraction interface of $\mathcal{F}_{\mathsf{AdaptSig}}$, which returns witness y.

Finally, the validation predicate $\mathsf{CheckAdapt}$ verifies the signatures attached to the transactions. Note that the instantiation of $\mathcal{G}_{\mathsf{Ledger}}$ used for $\Pi_{\mathsf{LedgerLocks}}$ differs from the one that $\mathcal{G}_{\mathsf{LedgerLocks}}$ extends. More specifically, $\mathcal{G}_{\mathsf{Ledger}}$ used in $\Pi_{\mathsf{LedgerLocks}}$ operates on transactions of the form $\mathtt{tx}^* = (\mathtt{tx}, \vec{\sigma})$ that (in addition to the account identities of $\mathtt{tx}$) hold the signatures $\vec{\mathsf{Sig}}$ that $\mathsf{CheckAdapt}$ verifies via $\mathcal{F}_{\mathsf{AdaptSig}}$. To hide this difference in format from a distinguishing environment, $\Pi_{\mathsf{LedgerLocks}}$ wraps the corresponding interfaces of $\mathcal{G}_{\mathsf{Ledger}}$ for reading and submitting.

5.2.3 Security Proof

The security of conditional ledger is captured with the following theorem.

Theorem 5. *The protocol $\Pi^{R}_{\mathsf{LedgerLocks}}$ UC-realizes $\mathcal{G}_{\mathsf{LedgerLocks}}$, in the $(\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid model.*

Proof. We note that $\mathcal{G}_{\mathsf{LedgerLocks}}$ mostly inherits the security properties of $\mathcal{G}_{\mathsf{Ledger}}$, since the only non-trivial properties that $\mathcal{G}_{\mathsf{LedgerLocks}}$ enforces (in addition to what the base ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ provides) are that only the account holders can submit transactions and transactions can be tied to conditions. Since both of these properties are achieved through the usage of adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$, we have that the real world indeed implements the stronger validation predicate. More precisely, due to the security of $\mathcal{F}_{\mathsf{AdaptSig}}$, we are guaranteed existence of the simulator $\mathcal{S}_{\mathsf{AdaptSig}}$ (as described in Section 4.2.3), which can handle our calls in ideal world execution to perfectly simulate the protocol. Our simulator $\mathcal{S}_{\mathsf{LedgerLocks}}$ is given below. We note that indistinguishability follows because the simulator $\mathcal{S}_{\mathsf{LedgerCond}}$ makes exactly the same calls to $\mathcal{F}_{\mathsf{AdaptSig}}$ that an honest party makes in $\Pi_{\mathsf{LedgerLocks}}$. Furthermore, in the case of releasing transactions, we have that the simulator $\mathcal{S}_{\mathsf{LedgerCond}}$ learns the witness as long as it is involved in the transaction, which coincides with the real world protocol. The rest of the operations reduces to the security of $\mathcal{G}_{\mathsf{Ledger}}$, which was proven to be secure and realizable in the UC framework by Badertscher et al. [BMTZ17]. Hence, as long as the protocol can be simulated in the ideal world, the ideal and real world executions are indistinguishable.

Simulator $\mathcal{S}_{\mathsf{LedgerLocks}}$

Initialization: The simulator internally runs $\mathcal{A}$ in a black-box way and simulates the interaction between $\mathcal{A}$ and (emulated) real-world hybrid functionalities. The inputs from $\mathcal{A}$ to the base ledger $\mathcal{G}_{\mathsf{Ledger}}$ are simply relayed (and replies given back to $\mathcal{A}$). The simulator maintains locally a list of keys $\mathcal{K}_P$, list of pre-signed transactions $\mathcal{P}_P$ and list of signed transactions $\mathcal{Q}_P$, for an honest party P. Moreover, the simulator has access to the adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$.

Messages from Lock-enabling Ledger:

- Upon receiving $(\mathsf{account\text{-}req}, \mathsf{sid}, (P', P))$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$, set $\mathsf{sid}' := (\mathsf{sid}, P, P')$, forward $(\mathsf{keygen}, \mathsf{sid}')$ to the simulated adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ in the name of P. Upon receiving $(\mathsf{verification\text{-}key}, \mathsf{sid}', \mathsf{vk})$ from $\mathcal{F}_{\mathsf{AdaptSig}}$, output this to $\mathcal{A}$ and store (P', vk) in $\mathcal{K}_P$.
- Upon receiving $(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}, \alpha)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$, parse α as $(\mathtt{AccountId}, \{P^*\})$ and $\{P^*\}$ as (P, P'), set $\mathsf{sid}' := (\mathsf{sid}, P, P')$, and forward $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, \perp, \mathit{signature})$ to the simulated adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ in the name of P. Upon receiving $(\mathsf{signature}, \mathsf{sid}', \sigma)$ from $\mathcal{F}_{\mathsf{AdaptSig}}$, output this answer to $\mathcal{A}$ and store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in list $\mathcal{Q}_P$.
- Upon receiving $(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}, \alpha, Y)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$, parse α as $(\mathtt{AccountId}, \{P^*\})$ and $\{P^*\}$ as (P, P'), set $\mathsf{sid}' := (\mathsf{sid}, P, P')$, forward $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, Y, \mathit{pre\text{-}signature})$ to the simulated adaptor signature functionality $\mathcal{F}_{\mathsf{AdaptSig}}$ in the name of P. Upon receiving $(\mathsf{signature}, \mathsf{sid}', \hat{\sigma})$ from $\mathcal{F}_{\mathsf{AdaptSig}}$, output this answer to $\mathcal{A}$ and store $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ in list $\mathcal{P}_P$.
- Upon receiving $(\mathsf{release\text{-}tx}, \mathsf{sid}, y)$ from $\mathcal{G}_{\mathsf{LedgerLocks}}$, store y.

This concludes the proof of Theorem 5. □

5.3 Template for Using LedgerLocks

We provide an overview of the LedgerLocks framework in Figure 5.6. The main purpose of LedgerLocks is to ease the description of an AS-based blockchain protocol Π_{App} and UC-realization of its corresponding ideal functionality $\mathcal{F}_{\mathsf{App}}$. More precisely, LedgerLocks allows modeling Π_{App} in $(\mathcal{G}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world and prove UC-realization of $\mathcal{F}_{\mathsf{App}}$ without the need to provide reductions to the properties of the adaptor signatures or condition creation mechanism.

We outline here in detail the steps towards using LedgerLocks framework for AS-based blockchain protocols. Later in Chapter 6 we apply these steps to showcase the usage of LedgerLocks with concrete applications of adaptor signatures.

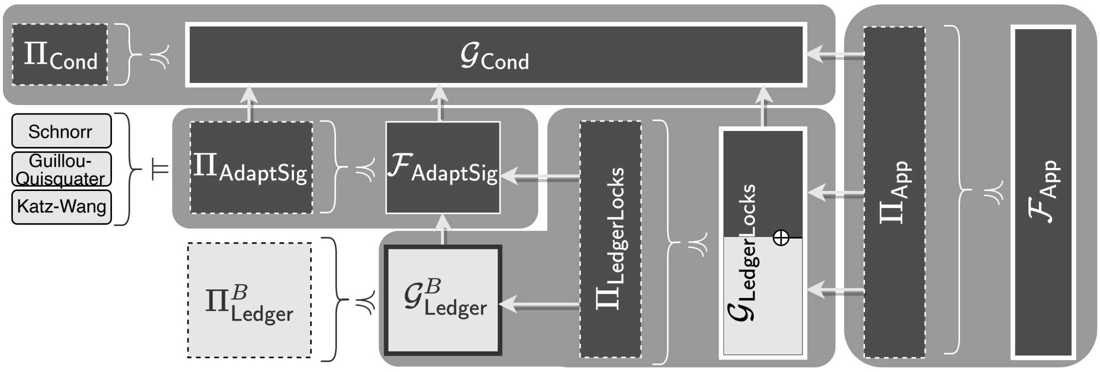

Π_{Cond}
$\preccurlyeq$
$\mathcal{G}_{\mathsf{Cond}}$
Schnorr
Guillou-Quisquater
Katz-Wang
$\models$
Π_{AdaptSig}
$\preccurlyeq$
$\mathcal{F}_{\mathsf{AdaptSig}}$
$\Pi_{\mathsf{LedgerLocks}}$
$\preccurlyeq$
$\mathcal{G}_{\mathsf{LedgerLocks}}$
$\oplus$
Π_{App}
$\preccurlyeq$
$\mathcal{F}_{\mathsf{App}}$
$\Pi^{B}_{\mathsf{Ledger}}$
$\preccurlyeq$
$\mathcal{G}^{B}_{\mathsf{Ledger}}$

Figure 5.6: Overview of the LedgerLocks framework.

Protocol Modeling. For describing a blockchain application protocol Π_{App} (such as the payment channel protocol Π_{Channel} given in Section 6.1) with the help of LedgerLocks, Π_{App} can be defined in a $(\mathcal{G}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world, meaning that it may interact with both $\mathcal{G}_{\mathsf{Cond}}$ and $\mathcal{G}_{\mathsf{LedgerLocks}}$. Intuitively, all (adaptor) signature-related operations of the protocols can be replaced with calls to the corresponding interfaces of $\mathcal{G}_{\mathsf{LedgerLocks}}$. $\mathcal{G}_{\mathsf{Cond}}$ allows for a logical separation between the creation of conditions and their usage to restrict transaction publication on $\mathcal{G}_{\mathsf{LedgerLocks}}$.

In cases where Π_{App} involves several blockchains, the description of Π_{App} uses multiple instances of $\mathcal{G}_{\mathsf{LedgerLocks}}$. The characteristics of these blockchain instances can be further refined by specifying the $\mathsf{CheckBase}$ predicate (and correspondingly the transaction format) to describe the logic of transaction execution. In Section 6.1 we show to instantiate $\mathsf{CheckBase}$ predicate to extend the blockchain logic with timelocks and UTXO style transactions. Though, we note that this formalism is expressive enough to encode more involved smart contract logic.

If Π_{App} requires the creation of conditions with additional properties (that go beyond the conditions presented in Section 4.1), then $\mathcal{G}_{\mathsf{Cond}}$ can be extended with additional condition creation interfaces to support the new condition types. In this case, for some relation R (known to realize $\mathcal{G}_{\mathsf{Cond}}$) one needs to give a protocol Π^{R}_{Cond} and prove that it realizes the newly added interfaces in $\mathcal{G}_{\mathsf{Cond}}$. Alternatively, one can immediately give a new protocol $\Pi^{R^*}_{\mathsf{Cond}}$ for some new relation R^* (that is known to be supported by some adaptor signature scheme) and show that it realizes $\mathcal{G}_{\mathsf{Cond}}$.

Defining Protocol Security. LedgerLocks can also serve as a starting point for defining the security of AS-based blockchain protocols. An ideal functionality $\mathcal{F}_{\mathsf{App}}$ capturing the desired security of Π_{App} can be described by extending $\mathcal{G}_{\mathsf{LedgerLocks}}$, e.g., by instantiating the $\mathsf{CheckBase}$ predicate that determines which transactions will be considered valid in a faithful protocol execution. Similar to how we extended $\mathcal{G}_{\mathsf{Ledger}}$ to $\mathcal{G}_{\mathsf{LedgerLocks}}$, $\mathcal{F}_{\mathsf{App}}$ may make use of additional state to capture the desired correctness and security properties of the protocol. For cross-chain blockchain protocols, $\mathcal{F}_{\mathsf{App}}$ may hold several internal copies of extended $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionalities.

Proving UC-Realization. Finally, one needs to prove that Π_{App} UC-realizes $\mathcal{F}_{\mathsf{App}}$. This proof should not involve cryptographic reductions[7] but focus on how the interactions of Π_{App} with $\mathcal{G}_{\mathsf{LedgerLocks}}$ and $\mathcal{G}_{\mathsf{Cond}}$ are translated into interactions with $\mathcal{F}_{\mathsf{App}}$. As $\mathcal{F}_{\mathsf{App}}$ uses $\mathcal{G}_{\mathsf{LedgerLocks}}$ as a component, the proof's essence should lie in showing that the way that transactions are created, locked and released within Π_{App} enforces the transaction inclusion logic encoded in $\mathcal{F}_{\mathsf{App}}$.

Limitations of LedgerLocks. LedgerLocks, in its current form, is not suitable for modeling blockchain protocols operating on ledgers that do not support transaction authorization through adaptor signature schemes[8]. However, most cryptocurrencies base their transaction authorization on signature schemes shown to support adaptor signatures, with one notable exception here being Zerocash [BCG$^+$14]. Furthermore, since $\mathcal{G}_{\mathsf{Ledger}}$ is currently only shown to be realized by the Bitcoin (Proof-of-Work) backbone protocol [BMTZ17] and the Ouroboros Genesis (Proof-of-Stake) protocol [BGK$^+$18], LedgerLocks (relying on the security of $\mathcal{G}_{\mathsf{Ledger}}$) only provides full end-to-end guarantees for ledgers using one of these protocols.

[7] Here we assume that Π_{App} does not make use of any other cryptographic primitive or protocol apart from adaptor signatures and condition creation protocol.

[8] We note that by doing minor modifications to $\mathcal{G}_{\mathsf{LedgerLocks}}$, our framework could be extended to also support ledgers without adaptor signature support if they implement a scripting language with native support for the condition checks (such as hash locks).

Ideal Functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$

The functionality interacts with an adversary $\mathcal{S}$ and a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$. It maintains a set of corrupted parties in $\mathcal{C}$. It uses $\mathcal{L}_{\mathsf{AccId}}$ with entries of the form $(\texttt{AccountId}, (P_1, \ldots, P_m))$, $\mathcal{L}_{\mathsf{Auths}}$ with entries of the form $(\texttt{AccountId}, \texttt{tx})$, and $\mathcal{L}_{\mathsf{TxsCond}}$ with entries of the form $(\mathsf{sid}, \texttt{tx}, Y, \{y, \bot\})$. Moreover, we inherit the **read** and **submit** interfaces from $\mathcal{G}_{\mathsf{Ledger}}$ of [BMTZ17] (cf. Section 2.4).

Account Generation: Upon receiving $(\mathsf{create\text{-}account}, \mathsf{sid}, (P_1, \ldots, P_m))$ from P do the following:

- For each P_i in $(P_1, \ldots, P_m)$: Send $(\mathsf{acc\text{-}req}, \mathsf{sid}, (P_1, \ldots, P_m, P))$ to P_i and receive $(\mathsf{acc\text{-}rep}, \mathsf{sid}, b_i)$. If any $b_i = 0$, then ignore the request.
- Send $(\mathsf{account\text{-}req}, \mathsf{sid}, (P_1, \ldots, P_m, P))$ to $\mathcal{S}$, and upon receiving a reply $(\mathsf{account\text{-}rep}, \mathsf{sid}, \texttt{AccountId})$, add $(\texttt{AccountId}, (P_1, \ldots, P_m, P))$ in $\mathcal{L}_{\mathsf{AccId}}$ and return $(\mathsf{create\text{-}account}, \mathsf{sid}, \texttt{AccountId})$ to all $P_1, \ldots, P_m$ and P.

Authorize TX: Upon receiving $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \texttt{tx}, \texttt{AccountId})$ from P, do the following:

- Extract the pair $\alpha := (\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\mathsf{AccId}}$. If it does not exist, then ignore the request.
- Set $\texttt{auth-flag} := 1$. For $P_i \in \{P^*\} \setminus \{P\}$: Send $(\mathsf{auth\text{-}req}, \mathsf{sid}, \texttt{tx}, \alpha)$ to P_i and $\mathcal{S}$, and receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_i)$ from P_i. If $b_i = 0$, set $\texttt{auth-flag} := 0$.
- If $\texttt{auth-flag} = 1$, store $(\texttt{AccountId}, \texttt{tx})$ in $\mathcal{L}_{\mathsf{Auths}}$.
- Return $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \texttt{auth-flag})$ to P.

Lock TX: Upon receiving $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \texttt{tx}, \texttt{AccountId}, Y)$ from P, do the following:

- Extract the pair $\alpha := (\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\mathsf{AccId}}$. If it does not exist, then ignore the request.
- Set $\texttt{lock-flag} := 1$. For $P_i \in \{P^*\} \setminus \{P\}$: Send $(\mathsf{lock\text{-}req}, \mathsf{sid}, \texttt{tx}, \alpha, Y)$ to P_i and $\mathcal{S}$, and receive $(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_i)$ from P_i. If $b_i = 0$, set $\texttt{lock-flag} := 0$.
- If $\texttt{lock-flag} = 1$, store $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ in $\mathcal{L}_{\mathsf{TxsCond}}$.
- Return $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \texttt{lock-flag})$ to P.

Release TX: Upon receiving $(\mathsf{release\text{-}tx}, \mathsf{sid}, \texttt{tx}, \texttt{AccountId}, Y, y)$ from some party P, do the following:

- Extract the pair $(\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\mathsf{AccId}}$. If it does not exist, then ignore the request.
- If $P \notin \{P^*\}$, then ignore the request.
- Extract the entry $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ from $\mathcal{L}_{\mathsf{TxsCond}}$. If it does not exist, then ignore the request.
- Invoke $\mathcal{G}^{R}_{\mathsf{Cond}}$ on input $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y, y))$ and receive $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b)$.
- If $b = 1$, then replace $(\texttt{AccountId}, \texttt{tx}, Y, \bot)$ with $(\texttt{AccountId}, \texttt{tx}, Y, y)$ in $\mathcal{L}_{\mathsf{TxsCond}}$, and store $(\texttt{AccountId}, \texttt{tx})$ in $\mathcal{L}_{\mathsf{Auths}}$.
- Invoke $(\mathsf{submit}, \mathsf{sid}, \texttt{tx})$. Moreover, if $\exists P \in \{P^*\}$, such that $P \in \mathcal{C}$, then send $(\mathsf{release\text{-}tx}, \mathsf{sid}, y)$ to $\mathcal{S}$.
- Return $(\mathsf{release\text{-}tx}, \mathsf{sid}, b)$ to P.

Signal Witness: Upon receiving $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \texttt{AccountId}, \texttt{tx}, Y)$ from party P, do the following:

- Extract the pair $(\texttt{AccountId}, \{P^*\})$ from $\mathcal{L}_{\mathsf{AccId}}$. If it does not exist, then ignore the request.
- If $P \notin \{P^*\}$, then ignore the request.
- Set $\mathsf{state}_i := \mathsf{state}|_{min\{\mathsf{pt}_\mathsf{P}, |\mathsf{state}|\}}$. Check if $\mathsf{inState}(\texttt{tx}, \mathsf{state}_i)$. Otherwise, ignore the request.
- Extract the entry $(\texttt{AccountId}, \texttt{tx}, Y, w)$ from $\mathcal{L}_{\mathsf{TxsCond}}$, where $w := y$ or $w := \bot$. Otherwise, ignore the request.
- Return $(\mathsf{signal\text{-}tx}, \mathsf{sid}, w)$ to P.

Validate Predicate: Our predicate $\mathsf{CheckCond}(\texttt{tx}, \mathsf{state})$ instantiates $\mathsf{isValidTx}(\texttt{tx}, \mathsf{state})$ from [BMTZ17] as follows:

- Parse $\texttt{tx} := (\mathbb{A}, \texttt{tx}')$. Then, for $\texttt{AccountId}_i \in \mathbb{A}$: Set $b_{1,i} := ((\texttt{AccountId}_i, \texttt{tx}) \in \mathcal{L}_{\mathsf{Auths}})$.
- Set $b_2 := \mathsf{CheckBase}(\texttt{tx}, \mathsf{state})$.
- Return $b_{1,1} \wedge \ldots \wedge b_{1,|\mathbb{A}|} \wedge b_2$.

Figure 5.4: Ideal functionality $\mathcal{G}_{\mathsf{LedgerLocks}}$. Here, pt_P is P's pointer into the state, as defined for $\mathcal{G}_{\mathsf{Ledger}}$ [BMTZ17]. Moreover, $\mathsf{inState}(\texttt{tx}, \mathsf{state}) := \exists B \in \mathsf{state},\ \texttt{tx} \in \mathsf{Blockify}^{-1}(B)$, where $\mathsf{Blockify}$ is a predicate to parse transactions into a block [BMTZ17].

Protocol $\Pi^{R}_{\mathsf{LedgerLocks}}$

Each party has a list $\mathcal{K}$ with entries (P, vk), a list $\mathcal{P}$ with entries $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$, and a list $\mathcal{Q}$ with entries $(\mathtt{tx}, \mathsf{vk}, \sigma)$.

Account Generation: Party P upon receiving $(\mathsf{create\text{-}account}, \mathsf{sid}, P')$ from $\mathcal{E}$:

- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P, P')$, send (sid') to P', and invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{keygen}, \mathsf{sid}')$. Receive $(\mathsf{verification\text{-}key}, \mathsf{sid}, \mathsf{vk})$ from $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$, and store (P', vk) in $\mathcal{K}$.
- Party P': Receive sid' from P. Invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{keygen}, \mathsf{sid}')$ and receive $(\mathsf{verification\text{-}key}, \mathsf{sid}, \mathsf{vk})$. Store (P, vk) in $\mathcal{K}$.

Authorize TX: Party P upon receiving $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}, P_0, P_1, \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Send $(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}, \{P_0, P_1\})$ to P_0 and P_1 and receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, f_b)$ from each P_b. If $f_b = 0$, abort.
- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P_0, P_1)$ and send $(\mathsf{sid}', \mathsf{vk})$ to P_0 and P_1.
- Party P_b (symmetrically party P_{1-b}): Receive $(\mathsf{sid}', \mathsf{vk})$ from P. Parse $\mathsf{sid}' := (\mathsf{sid}, P_b, P_{1-b})$. Extract (P_{1-b}, vk) from $\mathcal{K}$, and otherwise abort. Invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, \perp, \textit{signature})$ and receive $(\mathsf{signature}, \mathsf{sid}', \sigma)$. Store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in $\mathcal{Q}$, and send σ to P.
- Party P: Receive σ from P_b and P_{1-b}, store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in $\mathcal{Q}$.

Lock TX: Party P upon receiving $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, P_0, P_1, \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Send $(\mathsf{pre\text{-}auth\text{-}req}, \mathsf{sid}, \mathtt{tx}, \{P_0, P_1\})$ to P_0 and P_1 and receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, f_b)$ from each P_b. If $f_b = 0$, abort.
- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P_0, P_1)$ and send $(\mathsf{sid}', \mathsf{vk})$ to P_0 and P_1.
- Party P_b (symmetrically party P_{1-b}): Receive $(\mathsf{sid}', \mathsf{vk})$ from P. Parse $\mathsf{sid}' := (\mathsf{sid}, P_b, P_{1-b})$. Extract (P_{1-b}, vk) from $\mathcal{K}$, otherwise, abort. Invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{sign}, \mathsf{sid}', \mathtt{tx}, \mathsf{vk}, Y, \textit{pre-signature})$ and receive $(\mathsf{signature}, \mathsf{sid}', \hat{\sigma})$. Store $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ in $\mathcal{P}$, and send $\hat{\sigma}$ to P.
- Party P: Receive $\hat{\sigma}$ from P_b and P_{1-b}, and store $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ in $\mathcal{P}$.

Release TX: Party P upon receiving $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, y, P, P', \mathsf{vk})$ from $\mathcal{E}$:

- Party P: Compute $\mathsf{sid}' := (\mathsf{sid}, P, P')$, extract entry $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ from $\mathcal{P}$, invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{adapt}, \mathsf{sid}', \hat{\sigma}, \mathsf{vk}, y)$, receive $(\mathsf{adapted\text{-}signature}, \mathsf{sid}', \sigma)$, and store $(\mathtt{tx}, \mathsf{vk}, \sigma)$ in $\mathcal{Q}$.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{submit}, \mathsf{sid}, (\mathtt{tx}, \sigma))$.

Signal Witness: Party P upon receiving $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{tx}, Y, \mathsf{vk})$ from $\mathcal{E}$:

- Extract the entry $(\mathtt{tx}, \mathsf{vk}, Y, \hat{\sigma})$ from $\mathcal{P}$, otherwise abort.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{read}, \mathsf{sid})$ and receive the current state.
- Check if $\mathsf{inState}(((\mathsf{vk}_1, \ldots \mathsf{vk}_n), \mathtt{tx}'), (\sigma_1, \ldots, \sigma_n), \mathsf{state})$ and $\mathsf{vk}_i = \mathsf{vk}$ for some vk_i, otherwise abort.
- Invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{extract}, \sigma_i, \hat{\sigma}, \mathsf{vk})$, receive $(\mathsf{witness}, \mathsf{sid}, y)$ and return y.

Ledger Read: Party P upon receiving $(\mathsf{read}, \mathsf{sid})$ from $\mathcal{E}$, do the following:

- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{read}, \mathsf{sid})$ and receive the current state $\mathsf{state} := \mathsf{st}_1 || \ldots || \mathsf{st}_n$.
- Set $\mathsf{state}' := \mathsf{st}_1$. Extract $(\mathtt{tx}_1, (\sigma_{1,1}, \ldots, \sigma_{1,n})) || \ldots || (\mathtt{tx}_m, (\sigma_{m,1}, \ldots, \sigma_{m,n'}))$ for $\mathsf{st}_2, \ldots, \mathsf{st}_n$.
- Define new block content $\vec{x}' := \mathtt{tx}_1 || \ldots || \mathtt{tx}_m$. Set $\mathsf{state}' := \mathsf{state}' || \mathsf{Blockify}(\vec{x}')$ and return $(\mathsf{read}, \mathsf{sid}, \mathsf{state}')$.

Submit TX: Party P upon receiving $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx})$ from $\mathcal{E}$:

- Parse $\mathtt{tx} := ((\mathsf{vk}_1, \ldots, \mathsf{vk}_n), \mathtt{tx}')$ and check that each vk_i is in $\mathcal{K}$. Otherwise, ignore the request.
- Read the state from $\mathcal{G}_{\mathsf{Ledger}}$ as above. For each vk_i, extract the entry $(\mathtt{tx}, \mathsf{vk}_i, \sigma_i)$ from $\mathcal{Q}$. If any of them is missing, then abort.
- Invoke $\mathcal{G}_{\mathsf{Ledger}}$ on input $(\mathsf{submit}, \mathsf{sid}, (\mathtt{tx}, (\sigma_1, \ldots, \sigma_n)))$.

Validate Predicate: Our predicate $\mathsf{CheckAdapt}(\mathtt{tx}, \mathsf{state})$ instantiates $\mathsf{isValidTx}(\mathtt{tx}, \mathsf{state})$ in [BMTZ17] as follows:

- Parse $\mathtt{tx}^* := (((\mathsf{vk}_1, \ldots, \mathsf{vk}_n), \mathtt{tx}'), (\sigma_1, \ldots, \sigma_n))$. For each pair $(\mathsf{vk}_i, \sigma_i)$, invoke $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$ on input $(\mathsf{verify}, \mathsf{sid}, \mathtt{tx}, \sigma_i, \mathsf{vk}_i, \perp, \mathsf{signature})$, receive $(\mathsf{verified}, \mathsf{sid}, \mathtt{tx}, f_i)$, and set $b_{1,i} := f_i$.
- Set $b_2 := \mathsf{CheckBase}(((\mathsf{vk}_1, \ldots, \mathsf{vk}_n), \mathtt{tx}'), \mathsf{state})$ and return $b_{1,1} \wedge \ldots \wedge b_{1,n} \wedge b_2$.

Figure 5.5: Protocol $\Pi^{R}_{\mathsf{LedgerLocks}}$ in the $(\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}, \mathcal{G}_{\mathsf{Ledger}})$-hybrid world.

CHAPTER 6

Applications of Adaptor Signatures

In this chapter we demonstrate two applications of adaptor signatures. First, in Section 6.1 we show a payment channel construction, and then in Section 6.2 we build on the previous construction to create a payment channel hub (PCH). Both of these constructions are presented within the LedgerLocks framework that we previously described in Chapter 5.

6.1 Payment Channels

In this section, we modify the generalized channels construction of Aumayr et al. [AEE+21] and model it within our LedgerLocks framework. We clarify the modifications to the ideal functionality and the protocol given in [AEE+21] at a later part of this section, and first focus on how to properly instantiate $\mathcal{G}_{\mathsf{LedgerLocks}}$ for this application scenario.

In order to model the payment channel protocol we need to instantiate $\mathcal{G}_{\mathsf{LedgerLocks}}$ to support simple UTXO style transactions and also instantiate the $\mathsf{CheckBase}$ predicate accordingly. Towards this end, we first fix the transaction format. We refine $\mathsf{CheckBase}$ to account for absolute (transaction-level) timelocks by fixing the transaction format of $\mathtt{tx}$ to $(\mathtt{tx}', \mathit{tl})$, where tl denotes the *absolute timelock*. We further refine the format of $\mathtt{tx}'$ to be of the form $(\mathit{id}, \vec{\mathit{in}}, \vec{\mathit{out}})$ with id being a transaction identifier, $\vec{\mathit{in}}$ being a vector of inputs and $\vec{\mathit{out}}$ being a vector of outputs. Inputs $\mathit{in}_i \in \vec{\mathit{in}}$ are of the form $(\mathit{id}_{out}, j, \mathit{rtl})$, where (id_{out}, j) refers to the output that the input is spending (with id_{out} being the transaction id and j the offset in the transactions output vector), and rtl denotes a *relative timelock* indicating the number of blocks that need to have been included in the blockchain since the publication of the transaction with the referenced out before the transaction can be published. Outputs are $\mathit{out}_i \in \vec{\mathit{out}}$, and each one of them are of the form (aid, v), with aid denoting the id of the account controlling the output and v denoting the output's value.

In order to set up the timelocks correctly (which we discuss in more detail in Section 6.2), we define the parameter $\#_{safe}$ to be the maximum amount of time that it takes an honest party on the ledger to include a transaction in reaction to a change in the blockchain state. We set $\#_{safe} := 5 \cdot \texttt{windowsize}$, where $\texttt{windowsize}$ is a parameter that $\mathcal{G}_{\mathsf{LedgerLocks}}$ inherits from $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17], and denotes the maximum amount of blocks that an honest party can be lacking behind the current state of the blockchain. The reason for setting $\#_{safe} := 5 \cdot \texttt{windowsize}$ is that after transaction submission, $\mathcal{G}_{\mathsf{Ledger}}$ (and hence $\mathcal{G}_{\mathsf{LedgerLocks}}$) guarantees a valid transaction to appear in party P's view within $4 \cdot \texttt{windowsize}$ blocks.

Next, we refine the CheckBase predicate, denoted as $\mathsf{CheckBase}^{\mathbb{C}}$, to account for the additional UTXO checks. Intuitively, the following checks need to be conducted:

1. The transaction identifier is fresh.
2. The transaction inputs are unique.
3. The transactions should be required to be authorized by all accounts that control consumed inputs.
4. The values of the outputs created by a transaction should not exceed the values of the inputs consumed.
5. All consumed inputs should exist and respect the relative input timelocks.
6. All consumed inputs should not already been consumed by another transaction on the blockchain (i.e., no double-spending).

To simplify these checks, we define a helper function getOutput that accesses information of a transaction input in the blockchain state state. Given an input in and the blockchain state state, getOutput returns a set containing tuples with additional information for that output, namely the aid of the account controlling the spent output, the value v of the output and the height h at which the transaction with the output was added to the state.

Note that getOutput should return either $\emptyset$ indicating that state does not contain a transaction with the referred output or a singleton set containing the information for the unique output in state. Formally, getOutput is defined as follows.

$$\begin{aligned} &\mathsf{getOutput}((id_{out}, j, rtl), \mathsf{state}) := \\ &\qquad \{(id_{out}, j, aid, v, h) \mid \exists b, \mathsf{state}_{pre}, \mathsf{state}_{post}, \vec{in}, \vec{out}, \mathbb{A}, tl \text{ s.t.} \\ &\qquad\qquad \mathsf{state} = \mathsf{state}_{pre} || b || \mathsf{state}_{post} \\ &\qquad\qquad \wedge\ h = |\mathsf{state}_{pre}| \\ &\qquad\qquad \wedge\ (\mathbb{A}, ((id_{out}, \vec{in}, \vec{out}), tl) \in b \\ &\qquad\qquad \wedge\ out_j = (aid, v)\} \end{aligned}$$

Using $\mathsf{getOutput}$, we now define the $\mathsf{CheckBase}^{\mathbb{C}}$ predicate in Figure 6.1. The numbered conditions correspond to the checks described before.

$$
\begin{align*}
&\mathsf{CheckBase}^{\mathbb{C}}((\mathbb{A},(id,\vec{in},\vec{out})),\mathsf{state}) := \\
&\Big(\neg\exists\,\mathtt{tx}\ \mathbb{A}'\ id'\ \vec{in}'\ \vec{out}'\,tl', \mathtt{tx} = (\mathbb{A}',((id',\vec{in}',\vec{out}'),tl'))\ \land\ \mathsf{inState}(\mathtt{tx},\mathsf{state})\ \land\ id = id'\Big) \tag{1}\\
&\ \land\ \Big(\forall (id_{out},j,rtl)\ (id'_{out},j',rtl') \in \vec{in}.\ (id_{out},j) \neq (id'_{out},j')\Big) \tag{2}\\
&\ \land\ \left(\mathbb{A} = \left\{ aid \mid (id,j,aid,v,h) \in \bigcup_{in \in \vec{in}} \mathsf{getOutput}(in,\mathsf{state}) \right\}\right) \tag{3}\\
&\ \land\ \left(\sum_{(aid,v)\in\vec{out}} v \leq \sum_{(id',j',aid',v',h')\in\bigcup_{in\in\vec{in}}\mathsf{getOutput}(in,\mathsf{state})} v'\right) \tag{4}\\
&\ \land\ \Big(\forall (id_{out},j,rtl) \in \vec{in}.\ \exists\ aid'\ v'\ h'.\ \mathsf{getOutput}((id_{out},j,rtl),\mathsf{state}) = \{(id_{out},j,aid',v',h')\} \\
&\qquad \land\ |\mathsf{state}| - h \geq rtl\Big) \tag{5}\\
&\ \land\ \Big(\forall (id_{out},j,rtl) \in \vec{in}. \neg\exists\,\mathtt{tx}\ \mathbb{A}'\ id'\ \vec{in}'\ \vec{out}'\,tl'.\ \mathtt{tx} = (\mathbb{A}',((id',\vec{in}',\vec{out}'),tl'))\ \land\ \mathsf{inState}(\mathtt{tx},\mathsf{state}) \\
&\qquad \land\ \exists (id'_{out},j',rtl') \in \vec{in}'.\ (id'_{out},j') = (id_{out},j)\Big) \tag{6}
\end{align*}
$$

Figure 6.1: Definition of the $\mathsf{CheckBase}^{\mathbb{C}}$ predicate.

6.1.1 Ideal Functionality $\mathcal{G}_{\mathsf{Channel}}$

We capture the desired functionality of a payment channels as a global ideal functionality $\mathcal{G}_{\mathsf{Channel}}$. We model a payment channel γ as 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, $\gamma.\mathsf{users} \in \mathcal{P} \times \mathcal{P}$ denotes the parties involved in the channel. For convenience, we use $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{users} \to \gamma.\mathsf{users}$ defined as $\gamma.\mathsf{otherParty}(P) := Q$, for $\gamma.\mathsf{users} = \{P, Q\}$. Furthermore, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ represents the total amount of coins locked in γ, and $\gamma.\mathsf{st} = [\vec{\theta}_1, \ldots, \vec{\theta}_n]$ is the state of γ composed of a list of outputs. Each output $\vec{\theta}_i$ has two attributes (aid_i, v_i), where the value $v_i \in \mathbb{R}^{\geq 0}$ represents the amount of coins and the account identifier $aid_i \in \{0,1\}^*$ represents the spending account.

Before describing our functionality, we informally define below the properties of interest for our payment channels.

Consensus on creation: A payment channel γ is successfully created only if both parties in $\gamma.\mathsf{users}$ agree with the creation.

Consensus on update: A payment channel γ is successfully updated only if both parties in $\gamma.\mathsf{users}$ agree with the update.

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 $\gamma.\mathsf{st}$ 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).

Next, we describe our ideal functionality $\mathcal{G}_{\mathsf{Channel}}$, which is shown in Figure 6.2. The functionality keeps track of the corrupted parties in the set $\mathcal{C}$, Moreover, it also maintains a list Γ of all the created payment channels in their latest state and their corresponding funding transaction, and it is indexed by the channel identifier. Furthermore, $\mathcal{G}_{\mathsf{Channel}}$ makes calls to $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionality to read the latest ledger state.

The channel creation procedure is initiated by a party P sending a create message to $\mathcal{G}_{\mathsf{Channel}}$. At this point $\mathcal{G}_{\mathsf{Channel}}$ sends a channel creation request to $Q := \gamma.\mathsf{otherParty}(P)$, and if Q agrees (by sending back $\mathsf{create\text{-}ok}$ message), then $\mathcal{G}_{\mathsf{Channel}}$ expects a channel funding transaction, that spends both funding sources $\mathtt{tx}_P$ and $\mathtt{tx}_Q$, to appear on the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$. If this is true, then $\mathcal{G}_{\mathsf{Channel}}$ stores this transaction and the channel γ in Γ, and informs both parties of successful channel creation by sending them $\mathsf{created}$ message. We note that consensus on creation is achieved, since the channel creation is initiated by a party P, and only succeeds if Q agrees with it.

Similarly, channel closure can be requested by any $P \in \gamma.\mathsf{users}$ by sending a message close to $\mathcal{G}_{\mathsf{Channel}}$. Then, $\mathcal{G}_{\mathsf{Channel}}$ sends a channel closing request to $Q := \gamma.\mathsf{otherParty}(P)$, and if Q accepts the channel closure we have a peaceful closure. This in turn implies that a transaction that spends the channel funding transaction and whose output corresponds to the latest channel state $\gamma.\mathsf{st}$, should appear on the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$[9]. If only one party requests the channel closure or if one of the parties is adversarial, then we execute the $\mathtt{ForceClose}$ procedure, which ensures a forceful channel closing.

The channel update is initiated by one of the channel parties $P \in \gamma.\mathsf{users}$ by sending update message to $\mathcal{G}_{\mathsf{Channel}}$. Analogous to the generalized construction of Aumayr et al. [AEE$^+$21], the update is structured into two phases: (i) the prepare phase, and (ii) the revocation phase. At a high level, the prepare phase models the fact that both parties first agree on the new channel state, whereas the revocation phase models the fact that an update is only completed once the two parties invalidate the previous channel state. The parties P and Q agree to the prepare phase by sending $\mathsf{setup\text{-}ok}$ and $\mathsf{update\text{-}ok}$, respectively. Conversely, if these messages are not received it means that either of the parties does not agree on the update or setting up off-chain objects failed, in which case the parties abort the channel update. The abort can also result in forceful channel closing via a call to $\mathtt{ForceClose}$ procedure. The revocation phase is initiated by P sending revoke to $\mathcal{G}_{\mathsf{Channel}}$, at which point Q also needs to agree on the revocation by sending

[9] We note that we do not need to explicitly model the blockchain delay as this is taken care by $\mathcal{G}_{\mathsf{LedgerLocks}}$, which inherits these properties from $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17].

back $\mathsf{revoke\text{-}ok}$ message. At this point $\mathcal{G}_{\mathsf{Channel}}$ informs both parties of successful channel update by sending them $\mathsf{updated}$ message. Note that consensus on update is achieved as both parties need to agree on the channel update, and if parties are honest we have optimistic update procedure since update is independent of the ledger delay.

Ideal Functionality $\mathcal{G}_{\mathsf{Channel}}$

The functionality maintains a set of all parties $\mathcal{P}$, set of corrupted parties $\mathcal{C}$, s.t. $\mathcal{C} \subset \mathcal{P}$. Furthermore, we abbreviate $Q := \gamma.\mathsf{otherParty}(P)$ for $P \in \gamma.\mathsf{users}$ and $P \in \mathcal{P}$. The functionality also keeps track the payment channels in the list Γ, where $\Gamma(id)$ is a pointer to the channel γ with the identifier id, i.e., $\gamma.\mathsf{id} = id$.

Channel Creation: Upon receiving $(\mathsf{create}, \mathsf{sid}, \gamma, \mathtt{tx}_P, aid_P)$ from P, do the following:

- Send $(\mathsf{create\text{-}req}, \mathsf{sid}, \gamma)$ to Q.
- If received $(\mathsf{create\text{-}ok}, \mathsf{sid}, \gamma, \mathtt{tx}_Q, aid_Q)$ from Q, then invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input $(\mathsf{read}, \mathsf{sid})$, and receive back $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$. Otherwise, abort.
- If $\exists \mathtt{tx} \in \mathsf{state}$ s.t. $\mathtt{tx}.\vec{in} = (\mathtt{tx}_P.id, \mathtt{tx}_Q.id)$ and $\mathtt{tx}.\vec{out} = [(aid_{PQ}, \gamma.\mathsf{cash})]$, where $aid_{PQ} := (aid_P, aid_Q)$, then set $\Gamma(\gamma.\mathsf{id}) := (\{\gamma\}, \mathtt{tx})$ and send $(\mathsf{created}, \mathsf{sid}, \gamma)$ to $\gamma.\mathsf{users}$. Otherwise, abort.

Channel Update: Upon receiving $(\mathsf{update}, \mathsf{sid}, id, \vec{\theta})$ from P, do the following:

- Parse $(\{\gamma\}, \mathtt{tx}) := \Gamma(id)$ and set $\gamma' := \gamma$, $\gamma'.\mathsf{st} := \vec{\theta}$.
- Send $(\mathsf{update\text{-}req}, \mathsf{sid}, id, \vec{\theta})$ to Q and $(\mathsf{setup\text{-}req}, \mathsf{sid}, id)$ to P.
- If received $(\mathsf{setup\text{-}ok}, \mathsf{sid}, id)$ from P, then send $(\mathsf{setup\text{-}ok}, \mathsf{sid}, id)$ to Q. Otherwise, abort (P rejected).
- If received $(\mathsf{update\text{-}ok}, \mathsf{sid}, id)$ from Q, then send $(\mathsf{update\text{-}ok}, \mathsf{sid}, id)$ to P. Else, if $Q \notin \mathcal{C}$ (i.e., Q is not corrupted), then abort (Q rejected). Otherwise, set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathtt{tx})$, run $\mathtt{ForceClose}(id)$ and stop.
- If received $(\mathsf{revoke}, \mathsf{sid}, id)$ from P, then send $(\mathsf{revoke\text{-}req}, \mathsf{sid}, id)$ to Q. Else, set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathtt{tx})$, run $\mathtt{ForceClose}(id)$ and stop.
- If received $(\mathsf{revoke\text{-}ok}, \mathsf{sid}, id)$ from Q, then set $\Gamma(id) := (\{\gamma'\}, \mathtt{tx})$, send $(\mathsf{updated}, \mathsf{sid}, id, \vec{\theta})$ to $\gamma.\mathsf{users}$ and stop (accepted by both P and Q). Otherwise, set $\Gamma(id) := (\{\gamma, \gamma'\}, \mathtt{tx})$, run $\mathtt{ForceClose}(id)$ and stop.

Channel Closing: Upon receiving $(\mathsf{close}, \mathsf{sid}, id)$ from P, do the following:

- Send $(\mathsf{close\text{-}req}, \mathsf{sid}, id)$ to Q.
- If did not receive $(\mathsf{close\text{-}ok}, \mathsf{sid}, id)$ from Q or $P \in \mathcal{C} \vee Q \in \mathcal{C}$ (i.e., P or Q is corrupted), then run $\mathtt{ForceClose}(id)$ and stop.
- Otherwise, let $(\{\gamma\}, \mathtt{tx}) := \Gamma(id)$, invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input $(\mathsf{read}, \mathsf{sid})$, and receive back $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$.
- If $\exists \mathtt{tx}' \in \mathsf{state}$ s.t. $\mathtt{tx}'.\vec{in} = \mathtt{tx}.\vec{in}$ and $\mathtt{tx}'.\vec{out} = \gamma.\mathsf{st}$, set $\Gamma(id) := \perp$ and send $(\mathsf{closed}, \mathsf{sid}, id)$ to $\gamma.\mathsf{users}$. Otherwise, send $(\mathsf{error}, \mathsf{sid}, id)$ to $\gamma.\mathsf{users}$.

Monitor Status: Upon receiving $(\mathsf{monitor}, \mathsf{sid}, id)$ from P, do the following:

- If $\Gamma(id) = \perp$, then abort. Else, parse $(X, \mathtt{tx}) := \Gamma(id)$.
- Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input $(\mathsf{read}, \mathsf{sid})$, and receive back $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$.
- If $\exists \mathtt{tx}' \in \mathsf{state}$ s.t. $\mathtt{tx}'.\vec{in} = \mathtt{tx}.\vec{in}$ and $\mathtt{tx}'.\vec{out} = (aid_P, \gamma.\mathsf{cash})$, for $P \notin \mathcal{C}$, set $\Gamma(id) := \perp$, send $(\mathsf{punished}, \mathsf{sid}, id)$ to P and stop.

$\mathtt{ForceClose}(id)$ Let $\Gamma(id) := (X, \mathtt{tx})$. Invoke $\mathcal{G}_{\mathsf{LedgerLocks}}$ on input $(\mathsf{read}, \mathsf{sid})$, and receive back $(\mathsf{read}, \mathsf{sid}, \mathsf{state})$. If $\mathtt{tx} \in \mathsf{state}$ and $\mathtt{tx}$ unspent, then send $(\mathsf{error}, \mathsf{sid})$ to $\gamma.\mathsf{users}$ and abort.

Figure 6.2: The ideal functionality $\mathcal{G}_{\mathsf{Channel}}$.

The monitor interface allows the parties to monitor the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$ on request, in order to apply the punishment mechanism if misbehavior is detected. This is done by checking if there is a punish transaction on the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$, which spends the funding transaction of the channel γ and assigns $\gamma.\mathsf{cash}$ coins to the honest party $P \in \gamma.\mathsf{users}$. This ensures the instant finality with punishment property.

We note that our ideal functionality $\mathcal{G}_{\mathsf{Channel}}$ omits some natural checks that one would expect when receiving a message in order to ease the readability. For example, messages with missing parameters should be ignored, the channel instructions should only be accepted by parties of the channel, etc. One can formally capture these either by directly embedding them into the functionality or by defining a wrapper functionality around $\mathcal{G}_{\mathsf{Channel}}$, as done in [AEE+21].

Lastly, we remark that our functionality follows along the lines of the functionality given by Aumayr et al. [AEE+21], with a few differences which we clarify below.

1. We model the payment channels ideal functionality $\mathcal{G}_{\mathsf{Channel}}$ as a global functionality, as opposed to a local functionality as done in [AEE+21]. This is particularly motivated by the idea that once a payment channel is created it can serve as a building block for multiple other higher level protocols that utilize payment channels, such a payment channel network (PCN) construction using atomic multi-hop locks (AMHL) [MMS+19] or payment channel hub (PCH) construction using anonymous atomic lock (A2L) [TMM21a], which we present in Section 6.2. This implies that multiple protocols might need to share the same channel state, hence the need for a global functionality.

2. The ideal functionality of [AEE+21] is intended to be as generic as possible, and hence, it is parameterized by two values T and k. The value T is an upper bound on the maximal number of consecutive off-chain communication rounds between the parties, and the value k defines the number of ways that the channel state can be published on the ledger. On the other hand, our ideal functionality $\mathcal{G}_{\mathsf{Channel}}$ is not parameterized with any value. More precisely, we only consider the case of $k = 1$, which was also the case considered by Aumayr et al. [AEE+21] at a protocol level. Moreover, we do not keep track of the individual rounds and have no explicit upper bound T on the total off-chain communication rounds. Instead, we implicitly assume that there is some bound Δ, such that if the functionality is expecting a reply from a party, and it does not receive it within Δ rounds, then the functionality aborts the execution. This is without loss of generality, and is done only to make the functionality more readable.

3. Since the ideal functionality of [AEE+21] keeps track of the concrete rounds, at the end of each round the parties implicitly monitor their active payment channels in order to catch and punish the misbehaving parties. Our $\mathcal{G}_{\mathsf{Channel}}$ functionality instead provides an explicit monitor interface for this purpose, and the parties need to call explicitly this interface to punish the misbehaving parties.

4. The ideal functionality of [AEE+21] interacts with a very simplistic and ad-hoc ledger functionality, which is unclear if it can be realized, as discussed in Section 5.1. Instead, our $\mathcal{G}_{\mathsf{Channel}}$ functionality interacts with our $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionality, which in turn is based on the complete and realizable ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ of Badertscher et al. [BMTZ17], as described in Chapter 5.

6.1.2 Protocol Π_{Channel}

In this section we show how the generalized channels protocol of Aumayr et al. [AEE+21] can be described using our LedgerLocks framework. The protocol in [AEE+21] makes use of adaptor signatures, which in our case are encapsulated with AS-locked transactions in our LedgerLocks framework and $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionality.

Our protocol makes use of some auxiliary constructors for the transactions, which we show in Figure 6.3. The protocol Π_{Channel} is defined in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world and formally described in Figures 6.4 to 6.8.

We note that to shorten the description of the protocols and avoid repetition we omit the messages to and from the environment $\mathcal{E}$ and make use of the following arrow notation hereafter: by $m \hookrightarrow \mathcal{F}$ we mean "invoke the ideal functionality $\mathcal{F}$ on the input message m", and by $m \hookleftarrow \mathcal{F}$ we mean "receive the output message m from the ideal functionality $\mathcal{F}$".

Next, we describe the high-level details of the protocol. Our protocol follows the generalized channels protocol given by Aumayr et al. [AEE+21], with the exception that we make use of our LedgerLocks framework and $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionality, and we have a different punish mechanism, which we clarify below.

We note that the parties keep track of their channel state in Γ, which is indexed by the channel identifier. Hence, $\Gamma(id)$ returns the state of the channel γ, such that $\gamma.\mathsf{id} = id$. We can access any of the previous states of the channel by using the second index i, e.g., $\Gamma(id)[i]$ returns the i-th state of the channel γ, such that $\gamma.\mathsf{id} = id$. The total number of channel states is denoted by $|\Gamma(id)|$, hence, the index $i \in [1, |\Gamma(id)|]$. When the index i is omitted we simply return the latest channel state.

Channel Creation. The channel creation procedure is depicted in Figure 6.4. It starts by both parties generating their own pair of *revocation* public/secret pair (Y_R, y_R) and *publishing* public/secret pair (Y_P, y_P) and sharing with each other the public values Y_R, Y_P along with the funding transaction source (lines 1-4 of Figure 6.4). Next, they create a join account and construct the body of the funding, commit, split, punish and refund transactions (lines 5-12 of Figure 6.4). Analogous to the protocol of Aumayr et al. [AEE+21] the parties generate and exchange signatures on the split transaction $\mathtt{tx}_s$ which spends the commit transactions $\mathtt{tx}_c$ and whose output is equal to the initial state $\gamma.\mathsf{st} := [(aid_P, v_P), (aid_Q, v_Q)]$ and $\gamma.\mathsf{cash} := v_P + v_Q$. In our case this is handled with call to the $\mathsf{auth\text{-}tx}$ interface of $\mathcal{G}_{\mathsf{LedgerLocks}}$ (lines 23-24 of Figure 6.4). Similarly, the parties create a pre-signature on the commit transaction $\mathtt{tx}_c$ which is condition on the public value Y_P. This is handled in our case by making a call to the $\mathsf{lock\text{-}tx}$ interface of $\mathcal{G}_{\mathsf{LedgerLocks}}$ (lines 19-22 of Figure 6.4).

The difference with the protocol of Aumayr et al. [AEE$^+$21] is that for punish transaction we generate a merged condition $Y^* := f_{\mathsf{merge}}(stmt, R, Y_P, Y_R)$ and lock the transaction $\mathtt{tx}_p$ on Y^* (lines 13-18 and 25-28 of Figure 6.4). Lastly, the parties wait for the funding transaction $\mathtt{tx}_f$ on the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$ in $\#_{safe}$ time, and if yes, then the channel creation succeeds. Otherwise, the parties execute the refund procedure.

Channel Closing. The channel closing procedure is depicted in Figure 6.6. The purpose of the closing procedure is to collaboratively publish the latest channel state on the ledger. When parties want to close a channel, they first run a final update that 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 $\mathtt{tx}_f$ directly and sign this transaction (lines 3-5 of Figure 6.6). Next, they publish it on the ledger (line 11 of Figure 6.6). If the final update fails, parties close the channel forcefully (lines 15-16 of Figure 6.6).

Channel Update. To update a channel γ to a new state $\vec{\theta}$, given by a vector of outputs, parties have to (i) agree on the new commit and split transaction capturing the new state and (ii) invalidate the old commit transaction. The first part is similar to the agreement on the initial commit and split transaction as described previously in the creation protocol. To realize the second part, in which the punishment mechanism of the old commit transaction is activated, parties simply exchange the revocation secrets corresponding to the previous commit transaction (lines 26-30 of Figure 6.5). Moreover, analogous as in the channel creation procedure, we also create the body of the punish transaction $\mathtt{tx}_p$, generate a new merged condition Y^* and lock the punish transaction $\mathtt{tx}_p$ with the condition Y^* (lines 8-14 and 21-24 of Figure 6.5). This completes the honest channel update.

In the case of a misbehaving party, such as when one party locks (i.e., pre-signs) the new commit transaction $\mathtt{tx}_c$ and the other does not, or when one party revokes the old commit and the other does not, we instruct the protocol to execute a forceful closing procedure `ForceClose`.

Punish. If an honest party P detects that a malicious party Q posted an old commit transaction $\mathtt{tx}_c'$, it can react by publishing the punishing transaction $\mathtt{tx}_p^P$, which spends $\mathtt{tx}_c'$ and assigns all coins to P. In order to do so, party P needs to construct the secret y^{Q*}, which it needs to complete and release the punish transaction $\mathtt{tx}_p^P$. We note that $y^{Q*} := f_{\mathsf{merge}}(wit, R, y_P^Q, y_R^Q)$, and P has the revocation secret y_R^Q since the parties reveal their revocation secret to each other and $\mathtt{tx}_c'$ is old. Hence, P only needs to recover the secret y_P^Q in order to recover y^{Q*} and punish Q. Though, P can recover y_P^Q by making a call to the $\mathsf{signal\text{-}tx}$ interface of $\mathcal{G}_{\mathsf{LedgerLocks}}$ since the old transaction $\mathtt{tx}_c'$ has already be published on the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$. This procedure is shown in Figure 6.7.

GenFund($\texttt{tx}_P, \texttt{tx}_Q, aid_{PQ}, \gamma$)

```
parse $\texttt{tx}_P := (\mathbb{A}_P, ((id_P, \vec{in}_P, [(aid_P, v_P)]), tl_P))$
parse $\texttt{tx}_Q := (\mathbb{A}_Q, ((id_Q, \vec{in}_Q, [(aid_Q, v_Q)]), tl_Q))$
$id^* := H(id_P \| id_Q)$
return $((\{aid_P, aid_Q\}, ((id^*, [(id_P, 0, 0), (id_Q, 0, 0)], [(aid_{PQ}, \gamma.\mathsf{cash})]), 0)))$
```

GenRefund($\texttt{tx}_P, aid_R$)

```
parse $\texttt{tx}_P := (\mathbb{A}_P, ((id_P, \vec{in}_P, [(aid_P, v_P)]), tl_P))$
$id^* := H(id_P)$
return $((\{aid_R\}, ((id^*, [(id_P, 0, 0)], [(aid_R, v_P)]), 0)))$
```

GenCommit($\texttt{tx}_f$)

```
parse $\texttt{tx}_f := (\mathbb{A}, ((id, \vec{in}, [(aid_{PQ}, v)]), tl'))$
$id^* := H(id)$
return $(\{aid_{PQ}\}, ((id^*, [(id, 0, 0)], [(aid_{PQ}, v)]), 0))$
```

GenSplit($\texttt{tx}_c, \vec{\theta}$)

```
parse $\texttt{tx}_c := (\mathbb{A}, ((id, \vec{in}, [(aid_{PQ}, v)]), tl'))$
$id^* \leftarrow H(id)$
return $(\{aid_{PQ}\}, ((id^*, [(id, 0, \#_{safe})], \vec{\theta}), 0))$
```

GenPunish($\texttt{tx}_c, aid_P$)

```
parse $\texttt{tx}_c := (\mathbb{A}, ((id, \vec{in}, [(aid_{PQ}, v)]), tl'))$
$id^* := H(id)$
return $(\{aid_{PQ}\}, ((id^*, [(id, 0, 0)], [(aid_P, v)]), 0))$
```

GenPay($\texttt{tx}_s, aid_{PQ}, aid, tl$)

```
parse $\texttt{tx}_s := (\mathbb{A}, ((id, \vec{in}, [(aid_{PQ}, v), out_P, out_Q]), tl'))$
$id^* := H(id)$
return $(\{aid_{PQ}\}, ((id^*, [(id, 0, 0)], [(aid, v)]), tl))$
```

ComputeBalance($\texttt{tx}_f, \texttt{tx}_s$)

```
parse $\texttt{tx}_f := (\mathbb{A}_f, ((id_f, \vec{in}_f, \vec{out}_f), tl_f))$
parse $\texttt{tx}_s := (\mathbb{A}_s, ((id_s, \vec{in}_s, \vec{out}_s), tl_s))$
$id^* := H(id_f)$
return $(\{aid_{PQ}\}, ((id^*, [(id_f, 0, 0)], \vec{out}_s), 0))$
```

Figure 6.3: Definitions of transaction constructors used in the channel protocol.

Protocol Π_{Channel}

Create Channel

	$P(\gamma, \mathtt{tx}_P, aid_P)$		$Q(\mathtt{tx}_Q, aid_Q)$
1 :	$(Y^P_P, y^P_P) \leftarrow \mathsf{GenR}(1^\lambda)$		$(Y^Q_P, y^Q_P) \leftarrow \mathsf{GenR}(1^\lambda)$
2 :	$(Y^P_R, y^P_R) \leftarrow \mathsf{GenR}(1^\lambda)$		$(Y^Q_R, y^Q_R) \leftarrow \mathsf{GenR}(1^\lambda)$
3 :		$\xrightarrow{(\gamma, \mathtt{tx}_P, Y^P_P, Y^P_R)}$	
4 :		$\xleftarrow{(\mathtt{tx}_Q, Y^Q_P, Y^Q_R)}$	
5 :	$(\mathsf{create\text{-}account}, \mathsf{sid}, Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{acc\text{-}req}, \mathsf{sid}, (Q, P)) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
6 :			$(\mathsf{acc\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
7 :	$(\mathsf{create\text{-}account}, \mathsf{sid}, aid_{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{create\text{-}account}, \mathsf{sid}, aid_{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
8 :	$\mathtt{tx}_f := \mathit{GenFund}(\mathtt{tx}_P, \mathtt{tx}_Q, aid_{PQ}, \gamma)$		$\mathtt{tx}_f := \mathit{GenFund}(\mathtt{tx}_P, \mathtt{tx}_Q, aid_{PQ}, \gamma)$
9 :	$\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f)$		$\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f)$
10 :	$\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \gamma.\mathsf{st})$		$\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \gamma.\mathsf{st})$
11 :	$\mathtt{tx}^P_p := \mathit{GenPunish}(\mathtt{tx}_c, aid_P)$		$\mathtt{tx}^Q_p := \mathit{GenPunish}(\mathtt{tx}_c, aid_Q)$
12 :	$\mathtt{tx}^P_r := \mathit{GenRefund}(\mathtt{tx}_P, aid^r_P)$		$\mathtt{tx}^Q_r := \mathit{GenRefund}(\mathtt{tx}_Q, aid^r_Q)$
13 :	$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y^P_P, y^P_P)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y^Q_P, y^Q_P)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
14 :	$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y^P_P) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y^Q_P) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
15 :	$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y^P_R, y^P_R)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y^Q_R, y^Q_R)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
16 :	$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y^P_R) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y^Q_R) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
17 :	$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y^Q_P, Y^Q_R)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y^P_P, Y^P_R)) \hookrightarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
18 :	$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{Q*}) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$		$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{P*}) \hookleftarrow \mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}$
19 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y^P_P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y^Q_P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
20 :	$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y^Q_P) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y^P_P) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
21 :	$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b^P_c) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b^Q_c) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
22 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b^{PQ}_c) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b^{PQ}_c) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
23 :	$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}_s, (aid_{PQ}, (P, Q))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
24 :	$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b^Q_s) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b^Q_s, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
25 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}^P_p, aid_{PQ}, Y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}^Q_p, aid_{PQ}, Y^{P*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
26 :	$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}^Q_p, (aid_{PQ}, (P, Q)), Y^{P*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}^P_p, (aid_{PQ}, (P, Q)), Y^{Q*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
27 :	$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b^P_p) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b^Q_p) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
28 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b^Q_p) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b^P_p) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
29 :	**if** $b^Q_s \wedge b^{PQ}_c \wedge b^Q_p \neq 1$ **then abort**		**if** $b^{PQ}_c \wedge b^P_p \neq 1$ **then abort**
30 :	$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_f, aid_P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_f, aid_Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
31 :	$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
32 :	$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
33 :	$h_f := \lvert\mathsf{state}\rvert$		$h_f := \lvert\mathsf{state}\rvert$
34 :	$(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_f) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_f) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
35 :	**while** $\lvert\mathsf{state}\rvert < h_f + \#_{safe}$:		**while** $\lvert\mathsf{state}\rvert < h_f + \#_{safe}$:
36 :	$\quad(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$\quad(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
37 :	$\quad(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$\quad(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
38 :	**if** $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state})$ **then**		**if** $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state})$ **then**
39 :	$\quad\Gamma^P(\gamma.\mathsf{id}) := (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}^P_p,$		$\quad\Gamma^Q(\gamma.\mathsf{id}) := (\gamma, aid_Q, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}^Q_p,$
40 :	$\qquad\qquad Y^P_P, y^P_P, Y^Q_P, Y^P_R, y^P_R, Y^{Q*}, Y^Q_R)$		$\qquad\qquad Y^Q_P, y^Q_P, Y^P_P, Y^Q_R, y^Q_R, Y^{P*}, Y^P_R)$
41 :	**else**		**else**
42 :	$\quad(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}^P_r) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$\quad(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}^Q_r) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
43 :	$\quad$**while** $\lvert\mathsf{state}\rvert < h_f + 2 \cdot \#_{safe}$:		$\quad$**while** $\lvert\mathsf{state}\rvert < h_f + 2 \cdot \#_{safe}$:
44 :	$\qquad(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$\qquad(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
45 :	$\qquad(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$\qquad(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
46 :	$\quad$**if** $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state})$ **then**		$\quad$**if** $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state})$ **then**
47 :	$\qquad\Gamma^P(\gamma.\mathsf{id}) := (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}^P_p,$		$\qquad\Gamma^Q(\gamma.\mathsf{id}) := (\gamma, aid_Q, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}^Q_p,$
48 :	$\qquad\qquad\quad Y^P_P, y^P_P, Y^Q_P, Y^P_R, y^P_R, Y^{Q*}, Y^Q_R)$		$\qquad\qquad\quad Y^Q_P, y^Q_P, Y^P_P, Y^Q_R, y^Q_R, Y^{P*}, Y^P_R)$

Figure 6.4: Create channel protocol in $(\mathcal{G}^{R,f_{\mathsf{merge}}}_{\mathsf{Cond}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *GenFund*, *GenCommit*, *GenSplit*, and *GenPunish* denote the constructors for $\mathtt{tx}_f$, $\mathtt{tx}_c$, $\mathtt{tx}_s$ and $\mathtt{tx}_s$, respectively, as described in Figure 6.3.

Protocol Π_{Channel}

Update Channel

	$P(id, \vec{\theta})$		$Q(id, \vec{\theta})$
1 :	$(Y_P^P, y_P^P) \leftarrow \mathsf{GenR}(1^\lambda)$		$(Y_P^Q, y_P^Q) \leftarrow \mathsf{GenR}(1^\lambda)$
2 :	$(Y_R^P, y_R^P) \leftarrow \mathsf{GenR}(1^\lambda)$		$(Y_R^Q, y_R^Q) \leftarrow \mathsf{GenR}(1^\lambda)$
3 :		$\xrightarrow{(Y_P^P, Y_R^P)}$	
4 :		$\xleftarrow{(Y_P^Q, Y_R^Q)}$	
5 :	Extract $(aid_P, aid_{PQ}, \mathtt{tx}_f)$ from $\Gamma^P(id)$		Extract $(aid_Q, aid_{PQ}.\mathtt{tx}_f)$ from $\Gamma^Q(id)$
6 :	$\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f)$		$\mathtt{tx}_c := \mathit{GenCommit}(\mathtt{tx}_f)$
7 :	$\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \vec{\theta})$		$\mathtt{tx}_s := \mathit{GenSplit}(\mathtt{tx}_c, \vec{\theta})$
8 :	$\mathtt{tx}_p^P := \mathit{GenPunish}(\mathtt{tx}_c, aid_P)$		$\mathtt{tx}_p^Q := \mathit{GenPunish}(\mathtt{tx}_c, aid_Q)$
9 :	$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_P^P, y_P^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_P^Q, y_P^Q)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
10 :	$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y_P^P) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
11 :	$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_R^P, y_R^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_R^Q, y_R^Q)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
12 :	$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y_R^P) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y_R^Q) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
13 :	$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_P^Q, Y_R^Q)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_P^P, Y_R^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
14 :	$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{Q*}) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{P*}) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
15 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
16 :	$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y_P^P) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
17 :	$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_c^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_c^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
18 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_c^{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_c^{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
19 :	$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}_s, (aid_{PQ}, (P, Q))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
20 :	$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_s^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_s^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
21 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^P, aid_{PQ}, Y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^Q, aid_{PQ}, Y^{P*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
22 :	$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_p^Q, (aid_{PQ}, (P, Q)), Y^{P^*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_p^P, (aid_{PQ}, (P, Q)), Y^{P*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
23 :	$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_p^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_p^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
24 :	$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_p^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_p^P) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
25 :	**if** $b_s^Q \wedge b_c^{PQ} \wedge b_p^Q \neq 1$ **then abort**		**if** $b_c^{PQ} \wedge b_p^P \neq 1$ **then abort**
26 :	Extract $(\overline{Y_R^P}, \overline{y_r^P})$ from $\Gamma^P(id)$		Extract $(\overline{Y_R^Q}, \overline{y_r^Q})$ from $\Gamma^Q(id)$
27 :		$\xrightarrow{(\overline{Y_R^P}, \overline{y_R^P})}$	
28 :		$\xleftarrow{(\overline{Y_R^Q}, \overline{y_R^Q})}$	
29 :	$(\mathsf{open\text{-}cond}, \mathsf{sid}, (\overline{Y_R^Q}, \overline{y_R^Q})) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		$(\mathsf{open\text{-}cond}, \mathsf{sid}, (\overline{Y_R^P}, \overline{y_R^P})) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
30 :	$(\mathsf{opened\text{-}cond}, \mathsf{sid}, b_0^P) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		31 : $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b_0^Q) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
32 :	$b_1^P := \overline{Y_R^Q} \overset{?}{\in} \Gamma^P(id)$		$b_1^Q := \overline{Y_R^P} \overset{?}{\in} \Gamma^Q(id)$
33 :	**if** $b_0^P \wedge b_1^P \neq 1$ **then**		**if** $b_0^Q \wedge b_1^Q \neq 1$ **then**
34 :	**go to** $\mathtt{ForceClose}(id, \lvert\Gamma^P(id)\rvert - 1)$		**go to** $\mathtt{ForceClose}(id, \lvert\Gamma^Q(id)\rvert - 1)$
35 :	**else**		**else**
36 :	$\Gamma^P(id) := \Gamma^P(id) \cup (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P,$		$\Gamma^Q(id) := \Gamma^Q(id) \cup (\gamma, aid_Q, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^Q,$
37 :	$Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, \overline{y_R^Q})$		$Y_P^Q, y_P^Q, Y_P^P, Y_R^Q, y_R^Q, Y^{P*}, Y_R^P, \overline{y_R^P})$

Figure 6.5: Update channel protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *GenCommit*, *GenSplit* and *GenPunish* denote the constructors for $\mathtt{tx}_c$, $\mathtt{tx}_s$ and $\mathtt{tx}_p$, respectively as described in Figure 6.3.

Protocol Π_{Channel}

Close Channel

```
$P(id)$
parse $(\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P,$
        $Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q) := \Gamma^P(id)$
$\mathtt{tx}_t := ComputeBalance(\mathtt{tx}_f, \mathtt{tx}_s)$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_t, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_t) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $b_t = 0$ then
  go to $\mathtt{ForceClose}(id, |\Gamma^P(id)|)$
$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$h_t := |\mathsf{state}|$
$(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_t) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
while $|\mathsf{state}| < h_t + \#_{safe}$:
  $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $\neg\mathsf{inState}(\mathtt{tx}_t, \mathsf{state})$ then
  go to $\mathtt{ForceClose}(id, |\Gamma^P(id)|)$
else $\Gamma^P(id) := \bot$
```

```
$Q(id)$
parse $(\gamma, aid_Q, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P,$
        $Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{P*}, Y_R^P, y_R^P) := \Gamma^Q(id)$
$\mathtt{tx}_t := ComputeBalance(\mathtt{tx}_f, \mathtt{tx}_s)$
$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{tx}_t, (aid_{PQ}, (P, Q))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_t) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
```

Figure 6.6: Close channel protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Here, *ComputeBalance* denotes the constructor for $\mathtt{tx}_t$ as described in Figure 6.3.

Protocol Π_{Channel}

ForceClose Channel

```
$P(id, i)$
parse $(\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P,$
        $Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q) := \Gamma^P(id)[i]$
$(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^P, y_P^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{release\text{-}tx}, \mathsf{sid}, b := 1) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$h_c := |\mathsf{state}|$
while $|\mathsf{state}| < h_c + 2 \cdot \#_{safe}$:
  $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_s) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$\Gamma^P(id) := \bot$
```

Punish Channel

```
$P(id, i)$
parse $(\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P,$
        $Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q) := \Gamma^P(id)[i]$
$(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{signal\text{-}tx}, \mathsf{sid}, y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $y_P^Q = \bot$ then abort
else
  $y^{Q*} := f_{\mathsf{merge}}(wit, R, y_P^Q, y_R^Q)$
  $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^P, aid_{PQ}, Y^{Q*}, y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  $(\mathsf{release\text{-}tx}, \mathsf{sid}, b := 1) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  $\Gamma^P(id) := \bot$
```

Figure 6.7: ForceClose and Punish algorithms in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

Protocol Π_{Channel}

<u>Monitor Channel</u>

$P(id)$

```
while true:
    $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
    $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
    for $i \in [0, |\Gamma^P(id)| - 2]$:
Extract $\mathtt{tx}_c$ from $\Gamma^P(id)[i]$
        if $\mathsf{inState}(\mathtt{tx}_c, \mathsf{state})$ then
            go to $\mathsf{PunishChannel}(id, i)$
```

Figure 6.8: Monitor channel algorithm in $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world.

6.1.3 Security Proof

The security of the protocol Π_{Channel} is established with the following theorem.

Theorem 6. *The protocol Π_{Channel} UC-realizes $\mathcal{G}_{\mathsf{Channel}}$, in the $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid model.*

Proof. In order to prove the theorem we provide the code of the simulator that simulates the protocol Π_{Channel}, given access to the (global) ideal functionalities $\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$ and $\mathcal{G}_{\mathsf{LedgerLocks}}$. In our setting this is sufficient since parties do not obtain any secret inputs, but only receive commands from the environment $\mathcal{E}$, hence, we only need to handle different behavior of malicious parties. Moreover, our realization only depends on the external global ideal functionalities, and not on the security of any specific cryptographic primitive, therefore, we do not need any hybrid arguments. Due to these reasons as long as the protocol can be simulated in the ideal world, the ideal and real world executions are indistinguishable.

We describe below the case of P honest and Q corrupted, while the reverse case is symmetrical and simulation follows analogously.

Simulator for Channel Creation

<u>Case P is honest and Q is corrupted</u>

<u>Party P upon $(\mathsf{create}, \mathsf{sid}, \gamma, \mathsf{tid}_P) \hookleftarrow \mathcal{E}$:</u>

1. Set $id = \gamma.id$, send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, (R_P, r_P)) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_P, y_P)) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, and send $(id, \mathsf{tid}_P, R_P, Y_P) \hookrightarrow Q$.
2. Send $(\mathsf{create\text{-}account}, \mathsf{sid}, Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and then receive $(\mathsf{create\text{-}account}, \mathsf{sid}, aid_{PQ}) \hookleftarrow$

$\mathcal{G}_{\mathsf{LedgerLocks}}$.

3. Upon receiving $(id, \mathsf{tid}_Q, R_Q, Y_Q) \hookleftarrow Q$, create

$$\begin{aligned}[\mathtt{TX}_f] &:= \mathtt{GenFund}((\mathsf{tid}_P, \mathsf{tid}_Q), \gamma),\\ [\mathtt{TX}_c] &:= \mathtt{GenCom}([\mathtt{TX}_f]),\\ [\mathtt{TX}_s] &:= \mathtt{GenSplit}(\mathtt{tx}_c, \mathsf{txid}\|1, \gamma.\mathsf{st}).\end{aligned}$$

4. Send $(\mathsf{lock\text{-}tx}, \mathsf{sid}, [\mathtt{TX}_c], (aid_{PQ}, (P, Q)), Y_Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and $(\mathsf{auth\text{-}tx}, \mathsf{sid}, [\mathtt{TX}_s], aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.

1. Sample $(Y_P^P, y_P^P) \leftarrow \mathsf{GenR}(1^\lambda)$, $(Y_R^P, y_R^P) \leftarrow \mathsf{GenR}(1^\lambda)$, and send $(\gamma, \mathtt{tx}_P, Y_P^P, Y_R^P)$ to Q.
2. Upon receiving $(\mathtt{tx}_Q, Y_P^Q, Y_R^Q)$ from Q, send $(\mathsf{create\text{-}account}, \mathsf{sid}, Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and receive $(\mathsf{create\text{-}account}, \mathsf{sid}, aid_{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
3. Create the body of the funding, commit, split, punish and refund transactions:

$$\begin{aligned}\mathtt{tx}_f &:= \mathit{GenFund}(\mathtt{tx}_P, \mathtt{tx}_Q, aid_{PQ}, \gamma)\\ \mathtt{tx}_c &:= \mathit{GenCommit}(\mathtt{tx}_f)\\ \mathtt{tx}_s &:= \mathit{GenSplit}(\mathtt{tx}_c, \gamma.\mathsf{st})\\ \mathtt{tx}_p^P &:= \mathit{GenPunish}(\mathtt{tx}_c, aid_P)\\ \mathtt{tx}_r^P &:= \mathit{GenRefund}(\mathtt{tx}_P, aid_P^r)\end{aligned}$$

4. Send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_P^P, y_P^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_R^P, y_R^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, and $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_P^Q, Y_R^Q)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{Q*}) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$.
5. Send $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_c^P := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_c^{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
6. Send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, b_s^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
7. Send $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^P, aid_{PQ}, Y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_p^Q, (aid_{PQ}, (P, Q)), Y^{P*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_p^P := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_p^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
8. If $b_s^Q \wedge b_c^{PQ} \wedge b_p^Q \neq 1$, then abort. Otherwise, send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_f, aid_P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, set $h_f := |\mathsf{state}|$ and send $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_f) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
9. Wait for $|\mathsf{state}| \geq h_f + \#_{safe}$, obtain state from $\mathcal{G}_{\mathsf{LedgerLocks}}$, and if $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state}) = \mathsf{true}$, then set $\Gamma^P(\gamma.\mathsf{id}) := (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q)$.
10. Otherwise, send $(\mathsf{submit}, \mathsf{sid}, \mathtt{tx}_r^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, wait for $|\mathsf{state}| \geq h_f + 2 \cdot \#_{safe}$, and if if $\mathsf{inState}(\mathtt{tx}_f, \mathsf{state}) = \mathsf{true}$ then set $\Gamma^P(\gamma.\mathsf{id}) := (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q)$.

Simulator for Channel Update

Case P is honest and Q is corrupted

Upon P sending $(\mathsf{update}, \mathsf{sid}, id, \vec{\theta}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$, do the following:

1. Sample $(Y_P^P, y_P^P) \leftarrow \mathsf{GenR}(1^\lambda)$, $(Y_R^P, y_R^P) \leftarrow \mathsf{GenR}(1^\lambda)$, and send (Y_P^P, Y_R^P) to Q.
2. Upon receiving $(\mathsf{setup\text{-}req}, \mathsf{sid}, id) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$, extract $(aid_P, aid_{PQ}, \mathtt{tx}_f)$ from $\Gamma^P(id)$, and create the body of the commit, split and punish transactions:
$$\begin{aligned} \mathtt{tx}_c &:= \mathit{GenCommit}(\mathtt{tx}_f) \\ \mathtt{tx}_s &:= \mathit{GenSplit}(\mathtt{tx}_c, \vec{\theta}) \\ \mathtt{tx}_p^P &:= \mathit{GenPunish}(\mathtt{tx}_c, aid_P) \end{aligned}$$
3. Send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_P^P, y_P^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y_R^P, y_R^P)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, and $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y_P^Q, Y_R^Q)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, Y^{Q*}) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$.
4. If P sends $(\mathsf{setup\text{-}ok}, \mathsf{sid}, id) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$, then send $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_c, aid_{PQ}, Y_P^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_c, (aid_{PQ}, (P, Q)), Y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_c^P := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_c^{PQ}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
5. Send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{tx}_s, b_s^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
6. Send $(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{tx}_p^P, aid_{PQ}, Y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{tx}_p^Q, (aid_{PQ}, (P, Q)), Y^{P*}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_p^P := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_p^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
7. If $b_s^Q \wedge b_c^{PQ} \wedge b_p^Q \neq 1$, then instruct $\mathcal{G}_{\mathsf{Channel}}$ to abort, otherwise receive $(\mathsf{update\text{-}ok}, \mathsf{sid}, id) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$.
8. Upon receiving $(\mathsf{revoke}, \mathsf{sid}, id) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$, extract $(\overline{Y_R^P}, \overline{y_R^P})$ from $\Gamma^P(id)$, and send $(\overline{Y_R^P}, \overline{y_R^P})$ to Q.
9. Upon receiving $(\overline{Y_R^Q}, \overline{y_R^Q})$ from Q, send $(\mathsf{open\text{-}cond}, \mathsf{sid}, (\overline{Y_R^Q}, \overline{y_R^Q})) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, and receive $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b_0^P) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$.
10. Set $b_1^P := \overline{Y_R^Q} \overset{?}{\in} \Gamma^P(id)$, and if $b_0^P \wedge b_1^P \neq 1$, then execute $\mathtt{ForceClose}(id, |\Gamma^P(id)| - 1)$ and stop.
11. Set $\Gamma^P(id) := \Gamma^P(id) \cup (\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, \overline{y_R^Q})$.

Simulator for Channel Closing

Case P is honest and Q is corrupted

Upon P sending $(\mathsf{close}, \mathsf{sid}, id) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$, do the following:

1. Parse $(\gamma, aid_P, aid_{PQ}, \mathtt{tx}_f, \mathtt{tx}_c, \mathtt{tx}_s, \mathtt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q) := \Gamma^P(id)$.

2. Create the transaction $\texttt{tx}_t := \mathit{ComputeBalance}(\texttt{tx}_f, \texttt{tx}_s)$.
3. Send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \texttt{tx}_t, \mathit{aid}_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_t) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and if $b_t = 0$ execute $\texttt{ForceClose}(\mathit{id}, |\Gamma^P(\mathit{id})|)$.
4. Otherwise, send $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, set $h_f := |\mathsf{state}|$, and send $(\mathsf{submit}, \mathsf{sid}, \texttt{tx}_t) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
5. Wait for $|\mathsf{state}| \geq h_t + \#_{safe}$, and if $\mathsf{inState}(\texttt{tx}_t, \mathsf{state}) = \mathsf{true}$, then set $\Gamma^P(\mathit{id}) := \bot$. Otherwise, execute $\texttt{ForceClose}(\mathit{id}, |\Gamma^P(\mathit{id})|)$.

Simulator for Monitoring

Case P is honest and Q is corrupted

Upon P sending $(\mathsf{monitor}, \mathsf{sid}, \mathit{id}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$, then do the following:

1. Send $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
2. Parse $\Gamma^P(\mathit{id}) := \{(\gamma, \mathit{aid}_P, \mathit{aid}_{PQ}, \texttt{tx}_f, \texttt{tx}_c, \texttt{tx}_s, \texttt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q)\}_{i\in[m]}$, where $m := |\Gamma^P(\mathit{id})|$.
3. If $\mathsf{inState}(\texttt{tx}_c^{(i)}, \mathsf{state}) = \mathsf{true}$, for some $i \in [m]$, then execute punish channel as follows:
    a) Send $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \texttt{tx}_c, \mathit{aid}_{PQ}, Y_P^Q) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and receive $(\mathsf{signal\text{-}tx}, \mathsf{sid}, y_P^Q) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$. If $y_P^Q = \bot$, then abort, else set $y^{Q*} := f_{\mathsf{merge}}(\mathit{wit}, R, y_P^Q, y_R^Q)$.
    b) Send $(\mathsf{release\text{-}tx}, \mathsf{sid}, \texttt{tx}_p^P, \mathit{aid}_{PQ}, Y^{Q*}, y^{Q*}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, b := 1) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and set $\Gamma^P(\mathit{id}) := \bot$.

Simulator for $\texttt{ForceClose}^P(\boldsymbol{id})$

1. Parse $(\gamma, \mathit{aid}_P, \mathit{aid}_{PQ}, \texttt{tx}_f, \texttt{tx}_c, \texttt{tx}_s, \texttt{tx}_p^P, Y_P^P, y_P^P, Y_P^Q, Y_R^P, y_R^P, Y^{Q*}, Y_R^Q, y_R^Q) := \Gamma^P(\mathit{id})$.
2. Send $(\mathsf{release\text{-}tx}, \mathsf{sid}, \texttt{tx}_c, \mathit{aid}_{PQ}, Y_P^P, y_P^P) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, b := 1) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$. Send $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and set $h_c := |\mathsf{state}|$.
3. Wait for $|\mathsf{state}| \geq h_c + 2\cdot\#_{safe}$, then send $(\mathsf{submit}, \mathsf{sid}, \texttt{tx}_s) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and set $\Gamma^P(\mathit{id}) := \bot$.

This concludes the proof of Theorem 6. □

6.2 Payment Channel Hub

In this section we present a payment channel hub (PCH) construction, called Anonymous Atomic Locks (A^{2}L), which was initially proposed in [TMM21a], and later improved in [GMM$^+$22]. The high-level idea of the constructions given in [TMM21a, GMM$^+$22] is to use a *synchronization puzzle*.

A synchronization puzzle is a protocol between three parties: Alice, Bob, and Hub (refer to Figure 6.9 for a pictorial description). The synchronization puzzle begins with Hub and Bob executing a *puzzle promise* protocol (step 1) with respect to some message m_{HB}, such that Bob receives a puzzle τ that contains a hidden signature σ_{HB} on m_{HB}. Bob wishes to solve the puzzle and obtain the embedded signature σ_{HB}. To do this, Bob sends the puzzle τ privately to Alice (step 2), who executes a *puzzle solve* protocol (step 3) with Hub with respect to some other transaction m_{AH} such that, at the end of the protocol, Alice obtains the signature σ_{HB}, whereas Hub obtains a signature σ_{AH} on m_{AH}. Alice then sends the signature σ_{HB} privately to Bob (step 4). Such a protocol must satisfy the following properties.

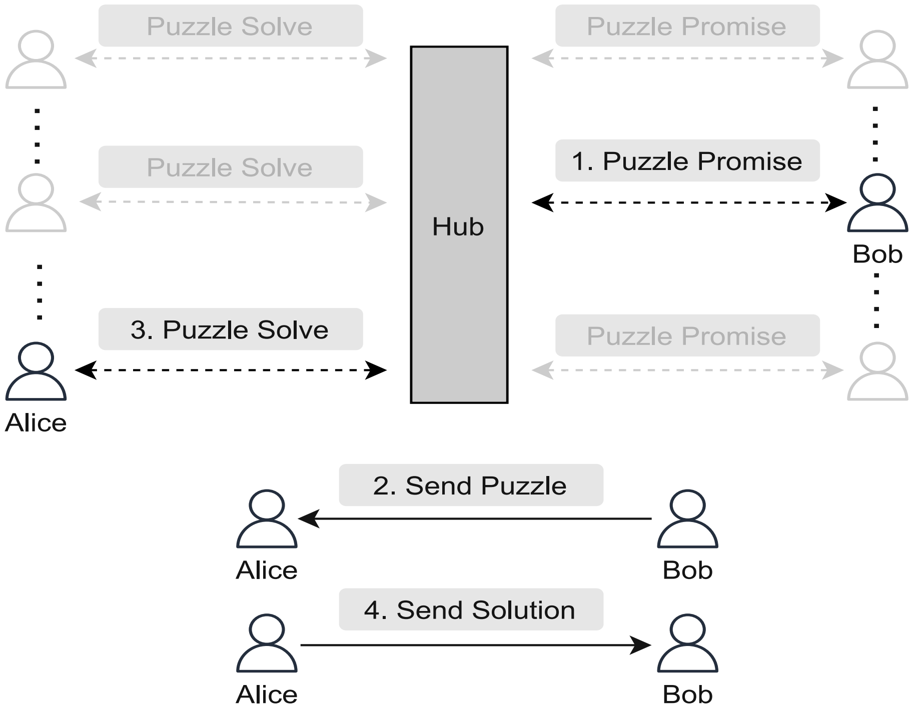


Figure 6.9: Protocol flow of the synchronization puzzle, the underlying cryptographic mechanism of TumbleBit [HAB$^+$17] and A^{2}L [TMM21a, GMM$^+$22]. Dotted double-edged arrows indicate 2-party protocols. Solid arrows indicate secure point-to-point communication.

Blindness: The puzzle solve protocol does not leak any information to Hub about τ, and Hub *blindly* helps solve the puzzle. This ensures that Hub cannot link puzzles across interactions.

Unlockability: If step 3 is successfully completed, then the secret s must be a valid secret for Bob's puzzle τ. This guarantees that Hub cannot learn a signature on m_{AH}, without at the same time revealing a signature on m_{HB}.

Unforgeability: Bob cannot output a valid signature on m_{HB} before Alice interacts with the Hub.

Towards a Payment Channel Hub. As shown in [HAB+17, TMM21a, GMM+22], a synchronization puzzle is the cryptographic core of a PCH. First, Alice and Bob define the messages as transactions, $m_{AH} := \mathtt{tx}_{AH}$ and $m_{HB} := \mathtt{tx}_{HB}$, as follows

$$\mathtt{tx}_{AH} : (A \xrightarrow{v} H) \text{ and } \mathtt{tx}_{HB} : (H \xrightarrow{v} B),$$

where $(P_i \xrightarrow{v} P_j)$ denotes a payment over payment channels that transfers v coins from party P_i to P_j. Second, Alice and Bob run the synchronization puzzle protocol with Hub to synchronize the two aforementioned transfers. Here, the signatures σ_{HB} and σ_{AH} are the ones required to validate the transactions $\mathtt{tx}_{HB}$ and $\mathtt{tx}_{AH}$, respectively. The anonymity of mixing follows from the fact that multiple pairs of users are executing the synchronization puzzle simultaneously with Hub, and Hub cannot link its interaction on the left to the corresponding interaction on the right.

In Section 6.2.1 we show an ideal functionality that captures the aforementioned properties and in Section 6.2.2 we show a concrete protocol realizing this ideal functionality. However, first we discuss the necessary timelocks, which are used in Section 6.2.2, i.e., at the protocol level.

Timelocks. As already discussed in Section 5.1, one of the most delicate points for protocol security are the concrete timelocks of the refund transactions and the corresponding reaction times of the participants in the protocol. For example, consider the previously discussed PCH scenario, where we note Hub with H, Alice with A and Bob with B. Here, the timelocks need to ensure that H can always claim the funds it pays to B from party A. To explain how the timelocks need to be set to ensure this, we consider the following worst-case scenario: Assume that B is not responding to H and that H at time t_{HB} (the timelock of its refund transaction $\mathtt{rtx}_{HB}$ for the money locked on the channel with B), wants to submit $\mathtt{rtx}_{HB}$. Then, H needs to consider that $\mathtt{rtx}_{HB}$ can only be published once the channel with B has been closed, meaning that both the commit transaction $\mathtt{tx}_c^{HB}$ and the split transaction $\mathtt{tx}_s^{HB}$ of this channel must have been published. Since publishing $\mathtt{tx}_c^{HB}$ takes up to $\#_{safe}$ block (from the perspective of H) and after that (due to its relative timelock) $\mathtt{tx}_s^{HB}$ can only be submitted after additional $\#_{safe}$ blocks and may take $\#_{safe}$ again until being published. Hence, H needs to start closing the channel at least at block height $t_{HB} - 3 \cdot \#_{safe}$ to be sure that $\mathtt{tx}_s^{HB}$ will be published at t_{HB} so that H can submit $\mathtt{rtx}_{HB}$.

Now, at this point, H cannot be sure that $\texttt{rtx}_{HB}$ will also be published on the blockchain since $\texttt{rtx}_{HB}$ can still be outrun by $\texttt{ctx}_{HB}$ (published by B). However, H is guaranteed that by $t_{HB} + \#_{safe}$ either $\texttt{rtx}_{HB}$ or $\texttt{ctx}_{HB}$ will be included in the ledger.

If indeed $\texttt{ctx}_{HB}$ was published, H still needs to have sufficient time to claim the payment from A. In the optimistic case, this can be settled by an off-chain channel update. However, if A does not collaborate, H also needs to close the channel with A (taking up to $3 \cdot \#_{safe}$ blocks), and afterwards publish $\texttt{ctx}_{AH}$ for claiming the money locked with A on this channel (taking other $\#_{safe}$ blocks). Consequently, it may take until $5 \cdot \#_{safe}$ blocks until H claims its funds in this way. To ensure that $\texttt{ctx}_{AH}$ is guaranteed to be published, the timelock t_{AH} of $\texttt{rtx}_{AH}$ needs to prevent that A could publish $\texttt{rtx}_{AH}$ before (and in this way outrun $\texttt{ctx}_{AH}$). For this reason, the parties need to ensure that $t_{AH} > t_{HB} + 5 \cdot \#_{safe}$. Therefore, in our protocols shown in Figures 6.15 and 6.16, we set $t_{HB} := 5 \cdot \#_{safe}$ and $t_{AH} := 2 \cdot t_{HB}$. Furthermore, the parties ensure that during the payment phase, they publish the claim transaction $\texttt{ctx}_{AH}$ at least $4 \cdot \#_{safe}$ before the timelock t_{AH} (so that it will be included before $\texttt{rtx}_{HB}$ is enabled).

Note that it is also crucial for security that an honest party H starts closing the channel with B (if B does not collaborate) latest at $t_{HB} - 3 \cdot \#_{safe}$ to make sure that at latest at $t_{HB} + \#_{safe}$, A knows whether she need to initiate the forceful claim. If this would be learned only later, the difference between the timelocks may not be sufficient to ensure a secure execution.

6.2.1 Ideal Functionality $\mathcal{F}_{\mathsf{A^2L}}$

We describe the ideal functionality $\mathcal{F}_{\mathsf{A^2L}}$ that captures the functionality and security that we except from a payment channel hub (PCH) construction using synchronization puzzles in the UC framework. Before describing our functionality, we informally define below the properties of interest.

Atomicity: Atomicity ensures that the receiver Bob only receives the coins, if some sender Alice paid the Hub first. Using our synchronization puzzle definition this means that a puzzle can only be solved, if there has been a corresponding execution of the puzzle solver protocol for that puzzle.

Unlinkability: Unlinkability means that the Hub does not learn any non-trivial information that allows it to associate the sender Alice and the receiver Bob of a payment (i.e, cannot link different runs of puzzle promise and puzzle solver protocols).

Griefing resistance: The PCH provides protection against *griefing attacks* [Rob19]. These attacks are mounted by Bob starting many puzzle promise operations, each of which requires Hub to lock coins, whereas the corresponding puzzle solver interactions are never carried out. As a consequence, all of Hub's coins are locked and no longer available, which results in a form of denial of service attack.

We note that the *blidness* property of a synchronization puzzle corresponds to the *unlinkability* property of a PCH. Moreover, the above *atomicity* guarantee implies the *unlockability* and *unforgeability* properties of a synchronization puzzle.

Our ideal functionality $\mathcal{F}_{\mathsf{A^2L}}$ is formally described in Figure 6.10. The functionality $\mathcal{F}_{\mathsf{A^2L}}$ manages a list $\mathcal{P}$ that keeps track of all the synchronization puzzles generated. The entries of the list $\mathcal{P}$ have the format (pid, b), where pid is the puzzle identifier, and b is a bit specifying whether the puzzle has already been solved or not. Additionally, $\mathcal{F}_{\mathsf{A^2L}}$ managed a list $\mathcal{T}$, which keeps track of the valid and used tokens.

The functionality $\mathcal{F}_{\mathsf{A^2L}}$ has four interfaces: registration, puzzle promise, puzzle solver, and open. The registration interface is initiated by the sender Alice by sending $\mathsf{register}$ to $\mathcal{F}_{\mathsf{A^2L}}$. At this point $\mathcal{F}_{\mathsf{A^2L}}$ sends a registration request to the Hub, and upon agreement by the Hub it generates a token tid which is send to both Alice and Bob. This token is used by the receiver Bob to initiate the puzzle promise by sending $\mathsf{promise}$ message along with the token tid to $\mathcal{F}_{\mathsf{A^2L}}$. Before processing the request, $\mathcal{F}_{\mathsf{A^2L}}$ ensures that the token is fresh and if not aborts the request. This ensures the authenticity of the requests, i.e., that puzzle promise can only be executed if a valid token has been acquired before and that each token can only be used once. This authenticity property allows to provide griefing protection if the sender Alice first has to lock a collateral before obtaining a token during the registration procedure. Although this does not fully block griefing attacks, it makes them financially prohibitive as the sender Alice has to lock coins for each execution of puzzle promise.

Inside the puzzle promise interface $\mathcal{F}_{\mathsf{A^2L}}$ sends a puzzle promise request to the Hub, and if it agrees, then it generates a fresh puzzle pid, which is stores inside $\mathcal{P}$ (indicating that it is currently unsolved) and shares the puzzle with Alice, Bob and the Hub. The puzzle solver phase is initiated by Alice sending solver message to $\mathcal{F}_{\mathsf{A^2L}}$. At that point $\mathcal{F}_{\mathsf{A^2L}}$ samples a new puzzle identifier pid' and asks the Hub if it wants to participate in the puzzle solver yes. This ensures unlinkability since the Hub only sees a fresh identifier pid' during the puzzle solver phase, which it cannot link to the original identifier pid that it received during puzzle promise phase. If the Hub agrees with the puzzle solving, then $\mathcal{F}_{\mathsf{A^2L}}$ marks the corresponding puzzle as solved in $\mathcal{P}$ and indicates this to Alice and Bob. This ensures atomicity since a puzzle is marked as solved only upon a successful execution of the puzzle solver procedure. Lastly, the open interface is initiated with open message to $\mathcal{F}_{\mathsf{A^2L}}$, and allows to check opening of the puzzle.

6.2.2 Protocol $\Pi_{\mathsf{A^2L}}$

The protocol $\Pi_{\mathsf{A^2L}}$ is defined in $(\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model, and formally described in Figures 6.13 to 6.16. Our construction makes use AS-locked transactions which are captured using $\mathcal{G}_{\mathsf{LedgerLocks}}$, along with cryptographic primitives such as a randomizable blind signature Σ_{RBS}, a commitment scheme Π_{COM}, a linearly homomorphic encryption scheme Π_{E}, a non-interactive zero-knowledge proof system Π_{NIZK} and a secure two-party computation (2PC) protocol Π_{2PC}.

Ideal Functionality $\mathcal{F}_{\mathsf{A^2L}}$

The functionality maintains a list $\mathcal{T}$ of tokens and list $\mathcal{P}$ of puzzles.

Registration: Upon receiving $(\mathsf{registration}, \mathsf{sid}, B)$ from A, do the following:

- Send $(\mathsf{registration\text{-}req}, \mathsf{sid}, A)$ to H.
- Receive $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b)$ from H.
- If $b = 0$, then abort. Else, sample $\mathsf{tid} \leftarrow_{\$} \{0,1\}^\lambda$ and add tid to $\mathcal{T}$.
- Send $(\mathsf{registered}, \mathsf{sid}, \mathsf{tid})$ to A, B and $\mathcal{S}$.

Puzzle Promise: Upon receiving $(\mathsf{promise}, \mathsf{sid}, \mathsf{tid}, A)$ from B, do the following:

- If $\mathsf{tid} \notin \mathcal{T}$, then abort. Otherwise, remove tid from $\mathcal{T}$.
- Send $(\mathsf{promise\text{-}req}, \mathsf{sid}, \mathsf{tid}, B)$ to H and $\mathcal{S}$.
- Receive $(\mathsf{promise\text{-}rep}, \mathsf{sid}, b)$ from H.
- If $b = 0$, then abort. Else, sample $\mathsf{pid} \leftarrow_{\$} \{0,1\}^\lambda$.
- Store the tuple $(\mathsf{pid}, \bot)$ into $\mathcal{P}$.
- Send $(\mathsf{promise}, \mathsf{sid}, \mathsf{pid})$ to A, B and H, and inform $\mathcal{S}$.

Puzzle Solver: Upon receiving $(\mathsf{solver}, \mathsf{sid}, B, \mathsf{pid})$ from A, do the following:

- If $(\mathsf{pid}, \bot) \notin \mathcal{P}$ then abort.
- Sample $\mathsf{pid}' \leftarrow_{\$} \{0,1\}^\lambda$.
- Send $(\mathsf{solve\text{-}req}, \mathsf{sid}, A, \mathsf{pid}')$ to H and $\mathcal{S}$.
- Receive $(\mathsf{solve\text{-}rep}, \mathsf{sid}, b)$ from H.
- If $b = 0$, then abort. Else, set the entry to $(\mathsf{pid}, \top)$ in $\mathcal{P}$.
- Send $(\mathsf{solved}, \mathsf{sid}, \mathsf{pid}, \top)$ to A, B and $\mathcal{S}$.

Open: Upon receiving $(\mathsf{open}, \mathsf{sid}, \mathsf{pid})$ from B, do the following:

- If $(\mathsf{pid}, b) \notin \mathcal{P}$ or $b = 0$, then send $(\mathsf{open}, \mathsf{sid}, \mathsf{pid}, b' := 0)$ to B and abort. Else, send $(\mathsf{open}, \mathsf{sid}, \mathsf{pid}, b' := 1)$ to B.

Figure 6.10: Ideal functionality $\mathcal{F}_{\mathsf{A^2L}}$.

We note that apart from the main protocols of $\Pi_{\mathsf{A^2L}}$, such as the registration, puzzle promise and puzzle solver protocols, we also define two auxiliary protocols for channel update and locking up coins based on a condition in Figures 6.11 and 6.12. These auxiliary protocols are defined in $(\mathcal{G}_{\mathsf{Channel}}, \mathcal{G}_{\mathsf{LedgerLocks}})$-hybrid world. Below we skip the auxiliary protocols and only explain the registration, puzzle promise and puzzle solver protocols.

Registration. The purpose of the registration protocol is to defend against the griefing attacks mentioned in Section 6.2.1. Our registration protocol is rather generic, and can be used with other constructions that require protection against similar type of griefing attack (e.g., TumbleBit [HAB+17]). The protocol is described in Figure 6.13.

The registration protocol is executed between the sender Alice and the Hub. It starts by Alice and Hub generating a joint account aid'_{AH}, and Alice locking coins in this escrow account (lines 1-13). Next, Alice samples a random token identifier tid and computes a commitment com to tid using the commitment scheme Π_{COM}, along with a NIZK proof π for the opening of the commitment, and sends the pair (com, π) to the Hub (lines 14-17). Then, the Hub verifies the proof π, and (blindly) generates a signature σ^* on the token

tid using the commitment com, and sends σ^* to Alice (lines 18-21). Here, it is important that tid is hidden (i.e., inside a commitment), otherwise, the Hub can trivially link Alice and Bob during the puzzle promise and puzzle solver phases. The reason for this is that the puzzle promise protocol (see Figure 6.15) starts with Bob sharing this tid in the clear with the Hub as a form of validation (i.e., that there already exists a payment promised to the Hub). This is also the reason why we require a signature scheme that allows to (blindly) sign a value hidden inside a commitment (such as Pointcheval-Sanders [PS16] signature scheme).

Next, Alice unblinds σ^* using the decommitment information decom to obtain a valid signature σ_{tid} on the token tid (line 22). Lastly, Alice sends the pair $(\mathsf{tid}, \sigma_{\mathsf{tid}})$ to Bob, which concludes the registration protocol.

Puzzle Promise. The puzzle promise protocol is depicted in Figure 6.15. It starts by Bob randomizing the signature σ_{tid} to obtain σ'_{tid} and sending the pair $(\mathsf{tid}, \sigma_{\mathsf{tid}})'$ to the Hub. We note that this randomization might not be necessary depending on the used signature scheme, because during the registration phase the Hub does not see the final signature. However, for example if one uses the Pointcheval-Sanders [PS16] signature scheme, then both blinded and unblinded signatures share one component that is the same and can be used by the Hub to break unlinkability. Hence, in this case we need to randomize the signature to break any link.

Once the Hub receives the pair $(\mathsf{tid}, \sigma_{\mathsf{tid}})'$, it checks that the signature $\sigma_{\mathsf{tid}})'$ is valid and that the token tid has not been previously used (lines 3-5), in order to be protected against replay attacks (i.e., Bob trying to claim the same collateral locked by Alice more than once). For this reason the Hub has to keep a list $\mathcal{T}$ of all the previously seen token.

Next, the Hub samples a statement/witness pair (Y, y), encrypts the witness y using the linearly homomorphic encryption scheme Π_{E} to obtain a ciphertext c, provides a NIZK proof π proving that the encrypted value satisfies is the witness of Y, i.e., $(Y, y) \in R$ and shares the tuple (Y, c, π) with Bob (lines 7-11). Here we have that the pair (Y, c) constitute the puzzle that Bob needs to solve in order to get paid by the Hub. More precisely, the next step involves the Hub and Bob setting up the commit and refund transactions, $\mathtt{ctx}_{HB}$ and $\mathtt{rtx}_{HB}$, respectively, by locking a payment from the Hub to Bob conditioned on the statement Y (lines 14-19). Since Bob cannot solve the puzzle himself, he forwards the puzzle (Y, c) to Alice, who can initiate the puzzle solver protocol with it in order to obtain the solution from the Hub.

We remark that the lines 20-24 from the Hub side of the puzzle promise protocol correspond to the Hub monitoring the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$ and executing the refund transaction $\mathtt{rtx}_{HB}$ in case the current state (i.e., block height) has passed $3 \cdot \#_{safe} + 1$ (setup of the timelocks was described at the beginning of this section). This only needs to happen if the peaceful update during the puzzle opening (Figure 6.14) fails.

Puzzle Solver. The puzzle solver protocol is shown in Figure 6.16. It starts by Alice generating a new statement/witness pair (Y, y') and creating a new merged condition of the form $Y^* := f_{\mathsf{merge}}(stmt, R, Y, Y')$, which she shares with the Hub (lines 1-6). This is done in order to randomize the puzzle and break any link with the previously run puzzle promise protocol.

Next, Alice and Hub setup the commit and refund transactions, $\texttt{ctx}_{AH}$ and $\texttt{rtx}_{AH}$, respectively, by locking a payment from Alice to Hub conditioned on the merged statement Y^* (lines 7-13). In order to unlock the transaction and get paid by Alice, the Hub needs to recover the witness y^*. This is done by Alice and Hub running a secure 2PC protocol Π_{2PC} [HK07], where Alice inputs the pair (y', c) and Hub inputs the decryption key dk_H. At the end of the protocol Hub learns the witness $y^* := f_{\mathsf{merge}}(wit, R, y^\dagger, y')$, where $y^\dagger$ is the decrypted value (line 14). If the extracted witness is correct, then the Hub shares it with Alice.

At this point, Alice and Hub perform a peaceful channel update (lines 19-20) that transfers the coins from Alice to the Hub. However, if the peaceful update does not happen because one of the parties is not responding, then they monitor the ledger $\mathcal{G}_{\mathsf{LedgerLocks}}$ and either performing the forceful payment or refund depending on the timelocks (lines 21-28).

Lastly, Alice unmerges the witness y^* in order to obtain the witness y that Bob needs in order to get paid by the Hub. Alice shares y with Bob, who performs the puzzle opening and get paid by the Hub (Figure 6.14). During the puzzle opening Bob tries to perform peaceful update with the Hub, and if this fails, then it uses y to forcefully receive the money from the Hub.

Protocol $\Pi_{\mathsf{A^2L}}$

Update Channel

$\mathsf{Update}\langle P(\gamma, \vec{\theta}), \cdot\rangle$	$\mathsf{Update}\langle \cdot, Q(\gamma, \vec{\theta})\rangle$
$(\mathsf{update}, \mathsf{sid}, \gamma.\mathsf{id}, \vec{\theta}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$	
$(\mathsf{update\text{-}req}, \mathsf{sid}, \gamma.\mathsf{id}, \vec{\theta}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$	$(\mathsf{setup\text{-}req}, \mathsf{sid}, \gamma.\mathsf{id}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$
$(\mathsf{setup\text{-}ok}, \mathsf{sid}, \gamma.\mathsf{id}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$	$(\mathsf{setup\text{-}ok}, \mathsf{sid}, \gamma.\mathsf{id}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$
$(\mathsf{update\text{-}ok}, \mathsf{sid}, \gamma.\mathsf{id}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$	$(\mathsf{update\text{-}ok}, \mathsf{sid}, \gamma.\mathsf{id}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$
$(\mathsf{revoke}, \mathsf{sid}, \gamma.\mathsf{id}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$	$(\mathsf{revoke\text{-}req}, \mathsf{sid}, \gamma.\mathsf{id}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$
	$(\mathsf{revoke\text{-}ok}, \mathsf{sid}, \gamma.\mathsf{id}) \hookrightarrow \mathcal{G}_{\mathsf{Channel}}$
$(\mathsf{updated}, \mathsf{sid}, \gamma.\mathsf{id}, \vec{\theta}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$	$(\mathsf{updated}, \mathsf{sid}, \gamma.\mathsf{id}, \vec{\theta}) \hookleftarrow \mathcal{G}_{\mathsf{Channel}}$
return $\perp$	**return** $\perp$

Figure 6.11: Update protocol in the $\mathcal{G}_{\mathsf{Channel}}$-hybrid world.

Protocol $\Pi_{\mathsf{A^2L}}$

Lock Channel

```
$\mathsf{Lock}\langle P(\gamma, \vec{\theta}, t, Y), \cdot\rangle$
$\mathsf{Update}\langle P(\gamma, \vec{\theta}), \cdot\rangle$
Extract $(aid_P, aid_Q, aid_{PQ}, \mathtt{tx}_s)$ from $\Gamma^P(\gamma.\mathsf{id})$
$\mathtt{ctx} := GenPay(\mathtt{tx}_s, aid_{PQ}, aid_Q, 0)$
$\mathtt{rtx} := GenPay(\mathtt{tx}_s, aid_{PQ}, aid_P, t)$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{rtx}, aid_{PQ}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_a) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{lock\text{-}tx}, \mathsf{sid}, \mathtt{ctx}, aid_{PQ}, Y) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{lock\text{-}tx}, \mathsf{sid}, b_l) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $b_a = 0 \vee b_l = 0$ then abort
return $(\mathtt{ctx}, \mathtt{rtx})$
```

```
$\mathsf{Lock}\langle \cdot, Q(\gamma, \vec{\theta}, t, Y)\rangle$
$\mathsf{Update}\langle \cdot, Q(\gamma, \vec{\theta})\rangle$
Extract $(aid_P, aid_Q, aid_{PQ}, \mathtt{tx}_s)$ from $\Gamma^Q(\gamma.\mathsf{id})$
$\mathtt{ctx} := GenPay(\mathtt{tx}_s, aid_{PQ}, aid_Q, 0)$
$\mathtt{rtx} := GenPay(\mathtt{tx}_s, aid_{PQ}, aid_P, t)$
$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{rtx}, (aid_{PQ}, (P, Q))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_a) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{lock\text{-}req}, \mathsf{sid}, \mathtt{ctx}, (aid_{PQ}, (P, Q)), Y) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{lock\text{-}rep}, \mathsf{sid}, b_l) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
return $(\mathtt{ctx}, \mathtt{rtx})$
```

Figure 6.12: Lock protocol in the $(\mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid world. Here, *GenPay* denotes the constructors for `ctx` and `rtx`, respectively, as described in Figure 6.3.

Protocol $\Pi_{\mathsf{A^2L}}$

Registration

```
$\mathsf{Registration}\langle A(\gamma_{AH}, \mathsf{pk}_H, \mathtt{tx}_s), \cdot\rangle$
$(\mathsf{create\text{-}account}, \mathsf{sid}, H) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{create\text{-}account}, \mathsf{sid}, aid'_{AH}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
Extract $(aid_A, aid_H)$ from $\Gamma^A(\gamma_{AH}.\mathsf{id})$
$t := 3 \cdot 5 \cdot \#_{safe} + 1$
$\mathtt{ctx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_H, 0)$
$\mathtt{rtx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_A, t)$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{rtx}, aid'_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_a) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $b_a = 0$ then return $\bot$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{ctx}, aid'_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_a) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $b_a = 0$ then return $\bot$
$\mathsf{tid} \leftarrow\!\!\$\ \mathbb{Z}_q$
$(\mathsf{com}, \mathsf{decom}) \leftarrow \Pi_{\mathsf{COM}}(1^\lambda, \mathsf{tid})$
$\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}(\mathsf{com}, (\mathsf{decom}, \mathsf{tid}))$
                    $\xrightarrow{\mathsf{com}, \pi}$
                    $\xleftarrow{\sigma^*}$
$\sigma_{\mathsf{tid}} := \Sigma_{\mathsf{RBS}}.\mathsf{UnblindSig}(\mathsf{decom}, \sigma^*)$
if $\Sigma_{\mathsf{RBS}}.\mathsf{Ver}(\mathsf{pk}_H, \sigma_{\mathsf{tid}}, \mathsf{tid}) \neq 1$ then
    return $\bot$
return $(\mathsf{tid}, \sigma_{\mathsf{tid}})$
```

```
$\mathsf{Registration}\langle \cdot, H(\gamma_{AH}, \mathsf{sk}_H, \mathsf{pk}_H, \mathtt{tx}_s)\rangle$
$(\mathsf{acc\text{-}req}, \mathsf{sid}, (A, H)) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{acc\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{create\text{-}account}, \mathsf{sid}, aid'_{AH}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
Extract $(aid_A, aid_H)$ from $\Gamma^H(\gamma_{AH}.\mathsf{id})$
$t := 3 \cdot 5 \cdot \#_{safe} + 1$
$\mathtt{ctx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_H, 0)$
$\mathtt{rtx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_A, t)$
$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{rtx}, (aid'_{AH}, (A, H))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_a) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{ctx}, (aid'_{AH}, (A, H))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_a) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
if $\Pi_{\mathsf{NIZK}}.\mathsf{V}(\pi, \mathsf{com}) \neq 1$ then
    return $\bot$
$\sigma^* \leftarrow \Sigma_{\mathsf{RBS}}.\mathsf{BlindSig}(\mathsf{sk}_H, \mathsf{com})$
return $\bot$
```

Figure 6.13: Registration protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model.

Protocol $\Pi_{\mathsf{A^2L}}$

Open

$\mathsf{Open}\langle H(\gamma_{HB}, \vec{\theta}_{HB}), \cdot\rangle$	$\mathsf{Open}\langle \cdot, B(\gamma_{HB}, \tau, y)\rangle$
	parse $\tau := (\vec{\theta}_{HB}, \mathtt{ctx}_{HB}, aid_{HB}, Y, c)$
$\vec{\theta}_{HB} := [(aid_H, v_H - v_{pay}), (aid_B, v_B + v_{pay})]$	$\vec{\theta}_{HB} := [(aid_H, v_H - v_{pay}), (aid_B, v_B + v_{pay})]$
$\mathsf{Update}\langle H(\gamma_{HB}, \vec{\theta}_{HB}), \cdot\rangle$	$\mathsf{Update}\langle \cdot, B(\gamma_{HB}, \vec{\theta}_{HB})\rangle$
	while true:
	$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
	$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
	if $\lvert\mathsf{state}\rvert \geq 4 \cdot \#_{safe} + 1$ **then**
	$(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{HB}, aid_{HB}, Y, y) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
	$(\mathsf{release\text{-}tx}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
return $\perp$	**return** b

Figure 6.14: Open protocol $(\mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model.

Protocol $\Pi_{\mathsf{A^2L}}$

Puzzle Promise

$\mathsf{PPromise}\langle H(\gamma_{HB}, \mathsf{pk}_H, \mathsf{ek}_H), \cdot\rangle$		$\mathsf{PPromise}\langle \cdot, B(\gamma_{HB}, \mathsf{ek}_H, \mathsf{tid}, \sigma_{\mathsf{tid}})\rangle$
		$\sigma'_{\mathsf{tid}} \leftarrow \Sigma_{\mathsf{RBS}}.\mathsf{RandSig}(\sigma_{\mathsf{tid}})$
	$\xleftarrow{\mathsf{tid}, \sigma'_{\mathsf{tid}}}$	
if $\mathsf{tid} \in \mathcal{T} \vee \Sigma_{\mathsf{RBS}}.\mathsf{Ver}(\mathsf{pk}_H, \sigma'_{\mathsf{tid}}, \mathsf{tid}) \neq 1$ **then**		
return $\perp$		
else $\mathcal{T} := \mathcal{T} \cup \{\mathsf{tid}\}$		
$(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$		
$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		
$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, Y) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$		
$c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, y)$		
$\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{P}((\mathsf{ek}_H, Y, c), y)$		
	$\xrightarrow{Y, c, \pi}$	
		if $\Pi_{\mathsf{NIZK}}.\mathsf{V}((\mathsf{ek}_H, Y, c), \pi) \neq 1$ **then**
		return $\perp$
Extract (aid_H, aid_B) from $\Gamma^H(\gamma_{HB}.\mathsf{id})$		Extract (aid_H, aid_B, aid_{HB}) from $\Gamma^B(\gamma_{HB}.\mathsf{id})$
$v_H := GetBalance(\Gamma^H(\gamma_{HB}.\mathsf{id}), aid_H)$		$v_H := GetBalance(\Gamma^B(\gamma_{HB}.\mathsf{id}), aid_H)$
$v_B := GetBalance(\Gamma^H(\gamma_{HB}.\mathsf{id}), aid_B)$		$v_B := GetBalance(\Gamma^B(\gamma_{HB}.\mathsf{id}), aid_B)$
$\vec{\theta}_{HB} := [(aid_{HB}, v_{pay}), (aid_H, v_H - v_{pay}), (aid_B, v_B)]$		$\vec{\theta}_{HB} := [(aid_{HB}, v_{pay}), (aid_H, v_H - v_{pay}), (aid_B, v_B)]$
$t_{HB} := 5 \cdot \#_{safe} + 1$		$t_{HB} := 5 \cdot \#_{safe} + 1$
$(\mathtt{ctx}_{HB}, \mathtt{rtx}_{HB}) \leftarrow \mathsf{Lock}\langle H(\gamma_{HB}, \vec{\theta}_{HB}, t_{HB}, Y), \cdot\rangle$		$(\mathtt{ctx}_{HB}, \mathtt{rtx}_{HB}) \leftarrow \mathsf{Lock}\langle \cdot, B(\gamma_{HB}, \vec{\theta}_{HB}, t_{HB}, Y)\rangle$
while true:		Send (Y, c) to A
$(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		**set** $\tau := (\vec{\theta}_{HB}, \mathtt{ctx}_{HB}, aid_{HB}, Y, c)$
$(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		
if $\lvert\mathsf{state}\rvert \geq 3 \cdot \#_{safe} + 1$ **then**		
$(\mathsf{submit}, \mathsf{sid}, \mathtt{rtx}_{HB}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$		
return $\perp$		**return** τ

Figure 6.15: Puzzle promise protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model.

Protocol $\Pi_{\mathsf{A^2L}}$

Puzzle Solver

```
$\mathsf{PSolver}\langle A(\gamma_{AH}, Y, c), \cdot\rangle$ | | $\mathsf{PSolver}\langle \cdot, H(\gamma_{AH}, \mathsf{dk}_H)\rangle$
$(Y', y') \leftarrow \mathsf{GenR}(1^\lambda)$
$(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y', y')) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
$(\mathsf{created\text{-}ind\text{-}cond}, \mathsf{sid}, Y') \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
$(\mathsf{create\text{-}merged\text{-}cond}, \mathsf{sid}, (Y, Y')) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
$(\mathsf{created\text{-}merged\text{-}cond}, \mathsf{sid}, Y^*) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
 | $\xrightarrow{Y^*}$ |
Extract $(aid_A, aid_H, aid_{AH})$ from $\Gamma^A(\gamma_{AH}.\mathsf{id})$ | | Extract $(aid_A, aid_H, aid_{AH})$ from $\Gamma^H(\gamma_{AH}.\mathsf{id})$
$v_A := GetBalance(\Gamma^A(\gamma_{AH}.\mathsf{id}), aid_A)$ | | $v_A := GetBalance(\Gamma^H(\gamma_{AH}.\mathsf{id}), aid_A)$
$v_H := GetBalance(\Gamma^A(\gamma_{AH}.\mathsf{id}), aid_H)$ | | $v_H := GetBalance(\Gamma^H(\gamma_{AH}.\mathsf{id}), aid_H)$
$v^* := v_{pay} + v_{fee}$ | | $v^* := v_{pay} + v_{fee}$
$\vec{\theta}_{AH} := [(aid_{AH}, v^*), (aid_A, v_A - v^*), (aid_H, v_H)]$ | | $\vec{\theta}_{AH} := [(aid_{AH}, v^*), (aid_A, v_A - v^*), (aid_H, v_H)]$
$t_{AH} := 2 \cdot 5 \cdot \#_{safe} + 1$ | | $t_{AH} := 2 \cdot 5 \cdot \#_{safe} + 1$
$(\mathtt{ctx}_{AH}, \mathtt{rtx}_{AH}) \leftarrow \mathsf{Lock}\langle A(\gamma_{AH}, \vec{\theta}_{AH}, t_{AH}, Y^*), \cdot\rangle$ | | $(\mathtt{ctx}_{AH}, \mathtt{rtx}_{AH}) \leftarrow \mathsf{Lock}\langle \cdot, H(\gamma_{AH}, \vec{\theta}_{AH}, t_{AH}, Y^*)\rangle$
 | $\Pi_{\mathsf{2PC}}((y', c), (\mathsf{dk}_H))$
 |   1 : if $\mathsf{ek}_H \neq \Pi_{\mathsf{E}}.\mathsf{Gen}(\mathsf{dk}_H)$
 |   2 :     then abort
 |   3 : $y^\dagger \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c)$
 |   4 : $y^* := f_{\mathsf{merge}}(wit, R, y^\dagger, y')$
 |   5 : return $((\bot), (y^*))$
 | | $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
 | | $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$
 | | if $b \neq 1$ then return $\bot$
 | $\xleftarrow{y^*}$ |
$\vec{\theta}_{AH} := [(aid_A, v_A - v^*), (aid_H, v_H + v^*)]$ | | $\vec{\theta}_{AH} := [(aid_A, v_A - v^*), (aid_H, v_H + v^*)]$
$\mathsf{Update}\langle A(\gamma_{AH}, \vec{\theta}_{AH}), \cdot\rangle$ | | $\mathsf{Update}\langle \cdot, H(\gamma_{AH}, \vec{\theta}_{AH})\rangle$
while true: | | while true:
  $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ | |   $(\mathsf{read}, \mathsf{sid}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ | |   $(\mathsf{read}, \mathsf{sid}, \mathsf{state}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  if $|\mathsf{state}| \geq 3 \cdot \#_{safe} + 1$ then | |   if $|\mathsf{state}| \geq 4 \cdot \#_{safe} + 1$ then
    $(\mathsf{submit}, \mathsf{sid}, \mathtt{rtx}_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ | |     $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{AH}, aid_{AH}, Y^*, y^*) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
  if $y^* = \bot$ then | |     $(\mathsf{release\text{-}tx}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
    $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{AH}, aid_{AH}, Y^*) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
    $(\mathsf{signal\text{-}tx}, \mathsf{sid}, y^*) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$
$y := f_{\mathsf{merge}}(wit, R, y^*, -y')$
Send $y$ to $B$
return $y$ | | return $\bot$
```

Figure 6.16: Puzzle solver protocol in $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model.

6.2.3 Security Proof

The security of the protocol $\Pi_{\mathsf{A^2L}}$ is established with the following theorem.

Theorem 7. *Let Σ_{RBS} be a* EUF-CMA *secure randomizable blind signature, Π_{COM} be a secure commitment scheme, Π_{E} be an* IND-CCA *secure encryption scheme with unique decryption keys, Π_{NIZK} be a UC-secure non-interactive zero-knowledge proof system, and Π_{2PC} be a UC-secure two-party computation protocol, then $\Pi_{\mathsf{A^2L}}$ UC-realizes $\mathcal{F}_{\mathsf{A^2L}}$, in the $(\mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}, \mathcal{G}_{\mathsf{LedgerLocks}}, \mathcal{G}_{\mathsf{Channel}})$-hybrid model.*

Proof. In order to prove the theorem we provide the code of the simulator that simulates the protocol $\Pi_{\mathsf{A^2L}}$, given access to the global ideal functionalities $\mathcal{G}_{\mathsf{Channel}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$ and $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$. We consider the cases where the adversary corrupts a different subset of parties separately. We describe the simulator for a single session and the security of the overall interaction is established via a standard hybrid argument, which reduces to the security of the underlying cryptographic primitives. We ignore the update and lock protocol given in Figures 6.11 and 6.12 in the rest of this proof since they are auxiliary protocols that only involve making calls to the $\mathcal{G}_{\mathsf{Channel}}$ and $\mathcal{G}_{\mathsf{LedgerLocks}}$ functionalities, hence, the simulator's job is trivial in this case.

H Corrupted. We give a simulator $\mathcal{S}_H$, then give a series of hybrid experiments that gradually change the real experiment (i.e., the construction in Figures 6.13, 6.15 and 6.16) into the ideal experiment given by the interaction of the corrupted H and the simulator $\mathcal{S}_H$, which has access to $\mathcal{F}_{\mathsf{A^2L}}$ along with the global ideal functionalities $\mathcal{G}_{\mathsf{Cond}}^{R,f_{\mathsf{merge}}}$, $\mathcal{G}_{\mathsf{LedgerLocks}}$ and $\mathcal{G}_{\mathsf{Channel}}$.

Simulator $\mathcal{S}_H$

Case A, B are honest and H are corrupted

Upon A sending $(\mathsf{registration}, \mathsf{sid}, B) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Send $(\mathsf{create\text{-}account}, \mathsf{sid}, H) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{create\text{-}account}, \mathsf{sid}, aid'_{AH}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
2. Extract (aid_A, aid_H) from $\Gamma^A(\gamma_{AH}.\mathsf{id})$, set $\mathtt{ctx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_H, 0)$ and $\mathtt{rtx} := GenPay(\mathtt{tx}_s, aid'_{AH}, aid_A, t)$, for $t := 3 \cdot 5 \cdot \#_{safe} + 1$.
3. Send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{rtx}, aid'_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_a) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and if $b_a = 0$, then return $\perp$.
4. Send $(\mathsf{auth\text{-}tx}, \mathsf{sid}, \mathtt{ctx}, aid'_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{auth\text{-}tx}, \mathsf{sid}, b_a) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and if $b_a = 0$, then return $\perp$.
5. Compute $(\mathsf{com}, \mathsf{decom}) \leftarrow \Pi_{\mathsf{COM}}(1^\lambda, 0)$, simulate the NIZK proof $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Sim}(\mathsf{td}, \mathsf{com})$, and send (com, π) to H.
6. Upon receiving σ^* from H, compute $\sigma_{\mathsf{tid}} := \Sigma_{\mathsf{RBS}}.\mathsf{UnblindSig}(\mathsf{decom}, \sigma^*)$, check if $\Sigma_{\mathsf{RBS}}.\mathsf{Ver}(\mathsf{pk}_H, \sigma_{\mathsf{tid}}, \mathsf{tid}) = 1$, then send $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$. Otherwise, send $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b := 0) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$.
7. Receive $(\mathsf{registered}, \mathsf{sid}, \mathsf{tid}) \hookleftarrow \mathcal{F}_{\mathsf{A^2L}}$ and store $(\mathsf{tid} := 0, \sigma_{\mathsf{tid}})$.
8. If at any point before the successful completion of the registration protocol, the adversary produces a valid signature σ_{tid} on any $\mathsf{tid} \notin \mathcal{T}$, then send $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b := 0) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and abort.

Upon B sending $(\mathsf{promise}, \mathsf{sid}, A) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Compute $\sigma'_{\mathsf{tid}} \leftarrow \Sigma_{\mathsf{RBS}}.\mathsf{RandSig}(\sigma_{\mathsf{tid}})$ and send $(\mathsf{tid}, \sigma'_{\mathsf{tid}})$ to H.
2. Upon receiving (Y, c, π) from H, if $\Pi_{\mathsf{NIZK}}.\mathsf{V}((\mathsf{ek}_H, Y, c), \pi) = 1$, then send $(\mathsf{promise\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$. Otherwise, send $(\mathsf{promise\text{-}rep}, \mathsf{sid}, b := 0) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and abort.

3. Extract (aid_H, aid_B, aid_{HB}) from $\Gamma^B(\gamma_{HB}.\mathsf{id})$, set $v_H := GetBalance(\Gamma^B(\gamma_{HB}.\mathsf{id}), aid_H)$, $v_B := GetBalance(\Gamma^B(\gamma_{HB}.\mathsf{id}), aid_B)$, set $\vec{\theta}_{HB} := [(aid_{HB}, v_{pay}), (aid_H, v_H - v_{pay}), (aid_B, v_B)]$ and $t_{HB} := 5 \cdot \#_{safe} + 1$, and run $(\mathtt{ctx}_{HB}, \mathtt{rtx}_{HB}) \leftarrow \mathsf{Lock}\langle \cdot, B(\gamma_{HB}, \vec{\theta}_{HB}, t_{HB}, Y)\rangle$.
4. Receive $(\mathsf{promise}, \mathsf{sid}, \mathsf{pid}) \hookleftarrow \mathcal{F}_{\mathsf{A^2L}}$ and store $(\mathtt{ctx}_{HB}, aid_{HB}, Y, c)$.

Upon A sending $(\mathsf{solver}, \mathsf{sid}, B, \mathsf{pid}) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Sample a pair $(Y^*, y^*) \leftarrow \mathsf{GenR}(1^\lambda)$, send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y^*, y^*)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, Y^*) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, and send Y^* to H.
2. Extract (aid_A, aid_H, aid_{AH}) from $\Gamma^A(\gamma_{AH}.\mathsf{id})$, set $v_A := GetBalance(\Gamma^A(\gamma_{AH}.\mathsf{id}), aid_A)$, $v_H := GetBalance(\Gamma^A(\gamma_{AH}.\mathsf{id}), aid_H)$, $v^* := v_{pay} + v_{fee}$, set state $\vec{\theta}_{AH} := [(aid_{AH}, v^*), (aid_A, v_A - v^*), (aid_H, v_H)]$, set timelock $t_{AH} := 2 \cdot 5 \cdot \#_{safe} + 1$, and run $(\mathtt{ctx}_{AH}, \mathtt{rtx}_{AH}) \leftarrow \mathsf{Lock}\langle A(\gamma_{AH}, \vec{\theta}_{AH}, t_{AH}, Y), \cdot\rangle$.
3. Initiate the 2PC protocol, and run the 2PC simulator $\mathcal{S}_{\Pi_{\mathsf{2PC}}}$ to recover its inputs dk_H. If $\mathsf{ek}_H \neq \Pi_{\mathsf{E}}.\mathsf{Gen}(\mathsf{dk}_H)$, program the output of 2PC to $\bot$, otherwise to y^*.
4. Receive y^* from H, and if $y^* = \bot$, then $(\mathsf{signal\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{AH}, aid_{AH}, Y^*) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$ and receive $(\mathsf{signal\text{-}tx}, \mathsf{sid}, y^*) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
5. Set $\vec{\theta}_{AH} := [(aid_A, v_A - v^*), (aid_H, v_H + v^*)]$ and run $\mathsf{Update}\langle A(\gamma_{AH}, \vec{\theta}_{AH}), \cdot\rangle$.
6. If the above update failed, then read state from $\mathcal{G}_{\mathsf{LedgerLocks}}$, and if $|\mathsf{state}| \geq 3 \cdot \#_{safe} + 1$, then $(\mathsf{submit}, \mathsf{sid}, \mathtt{rtx}_{AH}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
7. Upon receiving $(\mathsf{solved}, \mathsf{sid}, \mathsf{pid}, \top) \hookleftarrow \mathcal{F}_{\mathsf{A^2L}}$, compute $y \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c)$, send $(\mathsf{open}, \mathsf{sid}, \mathsf{pid}) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and return y.

Upon B sending $(\mathsf{open}, \mathsf{sid}, \mathsf{pid}) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Extract the stored $(\mathsf{pid}, (\mathtt{ctx}_{HB}, aid_{HB}, Y, y, c), \bot)$ from the list $\mathcal{P}$.
2. Set $\vec{\theta}_{HB} := [(aid_H, v_H - v_{pay}), (aid_B, v_B + v_{pay})]$ and run $\mathsf{Update}\langle \cdot, B(\gamma_{HB}, \vec{\theta}_{HB})\rangle$.
3. If above update failed, then read state from $\mathcal{G}_{\mathsf{LedgerLocks}}$. If $|\mathsf{state}| \geq 4 \cdot \#_{safe} + 1$, then send $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{HB}, aid_{HB}, Y, y) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, b) \hookleftarrow \mathcal{F}_{\mathsf{A^2L}}$ and send $(\mathsf{open}, \mathsf{sid}, b) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and return $\bot$.

Hybrid $\mathcal{H}_0$: This corresponds to the real protocol execution (Figures 6.13, 6.15 and 6.16).

Hybrid $\mathcal{H}_1$: Replace the honestly computed NIZK proof π (Figure 6.13, line 16) with a simulated proof.

Hybrid $\mathcal{H}_2$: Replace the commitment com (Figure 6.13, line 15) with a commitment to zero.

Hybrid $\mathcal{H}_3$: Abort if any valid signature σ_{tid} is received on any $\mathsf{tid} \notin \mathcal{T}$ before the registration protocol has successfully completed.

Hybrid $\mathcal{H}_4$: Simulate the 2PC protocol Π_{2PC} (Figure 6.16, line 14) and send the output y^* to H.

Hybrid $\mathcal{H}_5$: Sample $(Y^*, y^*) \leftarrow \mathsf{GenR}(1^\lambda)$ (i.e., remove lines 4-5 from Figure 6.16). If $\Pi_{\mathsf{E}}.\mathsf{Gen}(\mathsf{dk}_H) = \mathsf{ek}_H$, then send y^* to H. Otherwise, send $\perp$.

Lemma 10. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}$$

Proof. The indistinguishability follows directly from the zero-knowledge property of the NIZK proof system Π_{NIZK}. More precisely, assume towards a contradiction that $\mathcal{E}$ can distinguish the two executions with a non-negligible probability. We give a reduction to the zero-knowledge property of Π_{NIZK}. The reduction sets the statement $x := (\mathsf{com} := \Pi_{\mathsf{COM}}(1^\lambda, \mathsf{tid}))$ for a randomly sampled $\mathsf{tid} \in \mathbb{Z}_q$, and sends it to the zero-knowledge challenger, which responds with a proof π that is either an honest proof or a simulated proof. The reduction then acts as Alice in its interaction with $\mathcal{E}$, computing everything as in hybrid $\mathcal{H}_0$, except that it uses the proof π it received from the zero-knowledge challenger. At the end of the execution, based on $\mathcal{E}$'s guess, it outputs a bit to the challenger (0 if $\mathcal{E}$ guesses hybrid $\mathcal{H}_0$, and 1 otherwise), which will be correct with non-negligible advantage. However, this violates the zero-knowledge property of Π_{NIZK}, and hence, the two executions must be indistinguishable. □

Lemma 11. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}$$

Proof. Assume towards a contradiction that $\mathcal{E}$ can distinguish the two executions with a non-negligible probability. We give a reduction to the computational hiding security game of Π_{COM}. The reduction sets $m_0 := \mathsf{tid}$ and $m_1 := 0$, sends them to the computational hiding challenger, that responds with a commitment com. The reduction then acts as Alice in its interaction with $\mathcal{E}$, computing everything as in hybrid $\mathcal{H}_1$, except that it uses the commitment com it received from the computational hiding challenger. At the end of the execution, based on $\mathcal{E}$'s guess, it outputs a bit to the challenger (0 if $\mathcal{E}$ guesses hybrid $\mathcal{H}_0$, and 1 otherwise), which will be correct with non-negligible advantage. This violates the computational hiding property of Π_{COM}, so the two executions must be indistinguishable. □

Lemma 12. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}$$

Proof. Any distinguishing advantage implies a case in which the adversary $\mathcal{A}$ outputs some valid signature σ_{tid} for some message $\mathsf{tid} \notin \mathcal{T}$. This signature is a winning instance in the EUF-CMA experiment of the randomizable blind signature scheme Σ_{RBS}, but by assumption this only occurs with a negligible probability, and hence, the distinguishing advantage must be negligible. □

Lemma 13. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}$$

Proof. This indistinguishability follows directly from the UC-security of the 2PC protocol Π_{2PC}. □

Lemma 14. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}}$$

Proof. The uniqueness of the decryption key and correctness of Π_{E}, $\mathsf{ek}_H = \Pi.\mathsf{Gen}(\mathsf{dk}_H)$ implies that $\Pi.\mathsf{Dec}(\mathsf{dk}_H, \Pi(\mathsf{ek}_H, m)) = m$ for all m in the message space $\mathcal{M}$ of Π_{E}. Thus, the output y^* of the 2PC protocol Π_{2PC} is necessarily $y + y'$, where $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$, such that $c = \Pi_{\mathsf{E}}(\mathsf{ek}_H, y) \wedge (Y, y) \in R$ (this is guaranteed by the correctness of the NIZK proof system Π_{NIZK}). Since y' is uniformly random, y^* is identically distributed to $f_{\mathsf{merge}}(wit, R, y, y')$. The same holds for Y^* and $f_{\mathsf{merge}}(stmt, R, Y, Y')$. Furthermore, it holds that $(Y^*, y*) \in R$. □

Lemma 15. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_5,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F}_{\mathsf{A^2L}},\mathcal{S},\mathcal{E}}$$

Proof. $\mathcal{H}_5$ is identical to the ideal world. Moreover, since the transition from the real world (i.e., $\mathcal{H}_0$) to the ideal world (i.e., $\mathcal{H}_5$) is indistinguishable, it implies that in the ideal and real world are indistinguishable. □

A,B Corrupted. We give a simulator $\mathcal{S}_{AB}$ that interacts with $\mathcal{F}_{\mathsf{A^2L}}$ and show by a series of hybrids that our protocol is indistinguishable from an ideal experiment in which the corrupted parties interact with the simulator $\mathcal{S}_{AB}$.

Simulator $\mathcal{S}_{AB}$

Case H is honest and A, B are corrupted

Upon receiving $(\mathsf{registration\text{-}req}, \mathsf{sid}, A)$ from $\mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Receive $(\mathsf{acc\text{-}req}, \mathsf{sid}, (A, H)) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{acc\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and receive $(\mathsf{create\text{-}account}, \mathsf{sid}, aid'_{AH}) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
2. Extract (aid_A, aid_H) from $\Gamma^H(\gamma_{AH}.\mathsf{id})$, set $\mathtt{ctx} := \mathit{GenPay}(\mathtt{tx}_s, aid'_{AH}, aid_H, 0)$ and $\mathtt{rtx} := \mathit{GenPay}(\mathtt{tx}_s, aid'_{AH}, aid_A, t)$.
3. Receive $(\mathsf{auth\text{-}req}, \mathsf{sid}, \mathtt{rtx}, (aid'_{AH}, (A, H))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_a := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
4. Receive $(\mathsf{auth\text{-}rep}, \mathsf{sid}, \mathtt{ctx}, (aid'_{AH}, (A, H))) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, send $(\mathsf{auth\text{-}rep}, \mathsf{sid}, b_a := 1) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.

5. Upon receiving (com, π) from A, if $\Pi_{\mathsf{NIZK}}.\mathsf{V}(\pi, \mathsf{com}) = 1$, then compute the signature $\sigma^* \leftarrow \Sigma_{\mathsf{RBS}}.\mathsf{BlindSig}(\mathsf{sk}_H, \mathsf{com})$, send σ^* to A and $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$. Otherwise, send $(\mathsf{registration\text{-}rep}, \mathsf{sid}, b := 0) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and return $\bot$.

Upon receiving $(\mathsf{promise\text{-}req}, \mathsf{sid}, B)$ from $\mathcal{F}_{\mathsf{A^2L}}$, do the following:

1. Upon receiving $(\mathsf{tid}, \sigma'_{\mathsf{tid}})$ from B, if $\mathsf{tid} \in \mathcal{T} \vee \Sigma_{\mathsf{RBS}}.\mathsf{Ver}(\mathsf{pk}_H, \sigma'_{\mathsf{tid}}, \mathsf{tid}) \neq 1$, then send $(\mathsf{promise\text{-}rep}, \mathsf{sid}, b := 0) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$ and return $\bot$. Otherwise, set $\mathcal{T} := \mathcal{T} \cup \{\mathsf{tid}\}$ and send $(\mathsf{promise\text{-}rep}, \mathsf{sid}, b := 1) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$.
2. Sample a pair $(Y, y) \leftarrow \mathsf{GenR}(1^\lambda)$, send $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, (Y, y)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$ and receive $(\mathsf{create\text{-}ind\text{-}cond}, \mathsf{sid}, Y) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$.
3. Compute $c \leftarrow \Pi_{\mathsf{E}}.\mathsf{Enc}(\mathsf{ek}_H, 0)$, simulate the NIZK proof $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Sim}(\mathsf{td}, (\mathsf{ek}_H, Y, c))$ and send (Y, c, π) to B.
4. Extract (aid_H, aid_B) from $\Gamma^H(\gamma_{HB}.\mathsf{id})$, set $v_H := \mathit{GetBalance}(\Gamma^H(\gamma_{HB}.\mathsf{id}), aid_H)$, $v_B := \mathit{GetBalance}(\Gamma^H(\gamma_{HB}.\mathsf{id}), aid_B)$, set $\vec{\theta}_{HB} := [(aid_{HB}, v_{pay}), (aid_H, v_H - v_{pay}), (aid_B, v_B)]$ and $t_{HB} := 5 \cdot \#_{safe} + 1$, and run $(\mathtt{ctx}_{HB}, \mathtt{rtx}_{HB}) \leftarrow \mathsf{Lock}\langle H(\gamma_{HB}, \vec{\theta}_{HB}, t_{HB}, Y), \cdot \rangle$.
5. Read state from $\mathcal{G}_{\mathsf{LedgerLocks}}$, and if $|\mathsf{state}| \geq 3 \cdot \#_{safe} + 1$, then $(\mathsf{submit}, \mathsf{sid}, \mathtt{rtx}_{HB}) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$.
6. Upon receiving $(\mathsf{promise}, \mathsf{sid}, \mathsf{pid}) \hookleftarrow \mathcal{F}_{\mathsf{A^2L}}$, store $(\mathsf{pid}, (\mathtt{ctx}_{HB}, aid_{HB}, Y, y, c), \bot)$ into a list $\mathcal{P}$.

Upon receiving $(\mathsf{solve\text{-}req}, \mathsf{sid}, A, \mathsf{pid}')$ from $\mathcal{F}_{\mathsf{A^2L}}$ do the following:

1. Extract the stored $(\mathsf{pid}, (\cdot, \cdot, Y, y, c), \bot)$ from the list $\mathcal{P}$.
2. Upon receiving Y^* from A, extract (aid_A, aid_H, aid_{AH}) from $\Gamma^H(\gamma_{AH}.\mathsf{id})$, set values $v_A := \mathit{GetBalance}(\Gamma^H(\gamma_{AH}.\mathsf{id}), aid_A)$, $v_H := \mathit{GetBalance}(\Gamma^H(\gamma_{AH}.\mathsf{id}), aid_H)$, $v^* := v_{pay} + v_{fee}$, set state $\vec{\theta}_{AH} := [(aid_{AH}, v^*), (aid_A, v_A - v^*), (aid_H, v_H)]$, set timelock $t_{AH} := 2 \cdot 5 \cdot \#_{safe} + 1$, and run $(\mathtt{ctx}_{AH}, \mathtt{rtx}_{AH}) \leftarrow \mathsf{Lock}\langle \cdot, H(\gamma_{AH}, \vec{\theta}_{AH}, t_{AH}, Y) \rangle$.
3. Upon A initiating the 2PC protocol Π_{2PC}, run the 2PC simulator $\mathcal{S}_{\Pi_{\mathsf{2PC}}}$ to recover its inputs (y', c'), and program the output to $\bot$.
4. If $c' \in \mathcal{P}$, then set $y^* = f_{\mathsf{merge}}(wit, R, y, y')$, send $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$. If $b = 0$, then return $\bot$, otherwise, send y^* to A and $(\mathsf{solver}, \mathsf{sid}, B, \mathsf{pid}) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$.
5. If $c' \notin \mathcal{P}$, then compute $y^* \leftarrow \Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c') + y'$, send $(\mathsf{open\text{-}cond}, \mathsf{sid}, (Y^*, y^*)) \hookrightarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$, receive $(\mathsf{opened\text{-}cond}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{Cond}}^{R, f_{\mathsf{merge}}}$. If $b = 0$, then return $\bot$, otherwise, send y^* to A and send nothing to $\mathcal{F}_{\mathsf{A^2L}}$. (Note that this corresponds to the case where some party Alice is paying Hub without Bob initiating the interaction, which is something that she can do at any time.)
6. Set $\vec{\theta}_{AH} := [(aid_A, v_A - v^*), (aid_H, v_H + v^*)]$ and run $\mathsf{Update}\langle \cdot, H(\gamma_{AH}, \vec{\theta}_{AH}) \rangle$.
7. If above update failed, then read state from $\mathcal{G}_{\mathsf{LedgerLocks}}$. If $|\mathsf{state}| \geq 4 \cdot \#_{safe} + 1$, then send $(\mathsf{release\text{-}tx}, \mathsf{sid}, \mathtt{ctx}_{AH}, aid_{AH}, Y^*, y^*) \hookrightarrow \mathcal{G}_{\mathsf{LedgerLocks}}$, and receive $(\mathsf{release\text{-}tx}, \mathsf{sid}, b) \hookleftarrow \mathcal{G}_{\mathsf{LedgerLocks}}$. Send $(\mathsf{solve\text{-}rep}, \mathsf{sid}, b) \hookrightarrow \mathcal{F}_{\mathsf{A^2L}}$, and if $b = 1$, then set the last element of the above entry from $\mathcal{P}$ to $\top$ (to indicate that the puzzle promise is solved).

Hybrid $\mathcal{H}_0$: This corresponds to the real protocol execution (Figures 6.13, 6.15 and 6.16).

Hybrid $\mathcal{H}_1$: Replace the honestly computed NIZK proof π (Figure 6.15, line 10) with a simulated proof.

Hybrid $\mathcal{H}_2$: Simulate the 2PC protocol Π_{2PC} (Figure 6.16, line 14).

Hybrid $\mathcal{H}_3$: Add the list $\mathcal{P}$ and steps 4-5 of the simulator for puzzle solver to Figure 6.16, lines 15-17.

Hybrid $\mathcal{H}_4$: Replace the ciphertext c (Figure 6.15, line 9) with an encryption of zero.

Lemma 16. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_0,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}}$$

Proof. This indistinguishability follows directly from the zero-knowledge property of the NIZK proof system Π_{NIZK}, and the proof is analogous to that of Lemma 10. □

Lemma 17. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_1,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}}$$

Proof. This indistinguishability follows directly from the UC-security of the 2PC protocol Π_{2PC}. □

Lemma 18. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_2,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}}$$

Proof. By definition, for $c' \in \mathcal{P}$, the corresponding y and Y in $\mathbb{P}$ are $\Pi_{\mathsf{E}}.\mathsf{Dec}(\mathsf{dk}_H, c')$ and $(Y, y) \in R$, respectively. Therefore, we have that $y^* = f_{\mathsf{merge}}(wit, R, y, y')$, and the case of $c' \notin \mathcal{P}$ is handled in an analogous way. □

Lemma 19. *For all* PPT *distinguishers* $\mathcal{E}$*,*

$$\mathsf{EXEC}_{\mathcal{H}_3,\mathcal{A},\mathcal{E}} \approx \mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}}$$

Proof. Assume towards a contradiction that $\mathcal{E}$ can distinguish the two executions with a non-negligible probability. We give a reduction to the IND-CCA security game of Π_{E}. The reduction sets $m_0 := y$ and $m_1 := 0$, sends them to the IND-CCA challenger, that responds with c. The reduction then acts as the Hub in its interaction with $\mathcal{E}$, computing everything as in hybrid $\mathcal{H}_3$, except that it uses the ciphertext c it received from the IND-CCA challenger. Whenever it needs to decrypt some c^* it uses the IND-CCA decryption oracle. At the end of the execution, based on $\mathcal{E}$'s guess, it outputs a bit to the IND-CCA challenger (0 if $\mathcal{E}$ guesses hybrid $\mathcal{H}_3$, and 1 otherwise), which will be correct with non-negligible advantage. This violates the IND-CCA security of Π_{E}, so the two executions must be indistinguishable. □

Lemma 20. *For all* PPT *distinguishers* $\mathcal{E}$,

$$\mathsf{EXEC}_{\mathcal{H}_4,\mathcal{A},\mathcal{E}} \equiv \mathsf{EXEC}_{\mathcal{F}_{\mathsf{A^2L}},\mathcal{S},\mathcal{E}}$$

Proof. $\mathcal{H}_4$ is identical to the ideal world. Moreover, since the transition from the real world (i.e., $\mathcal{H}_0$) to the ideal world (i.e., $\mathcal{H}_4$) is indistinguishable, it implies that in the ideal and real world are indistinguishable. □

A,H Corrupted. This case is trivial, as Bob has no secret information and the simulator therefore simply follows the protocol.

H,B Corrupted. The simulator in this case follows the protocol honestly, and the security follows from the security of the AS-locked transactions, which is captured by $\mathcal{G}_{\mathsf{LedgerLocks}}$ and its usage of $\mathcal{F}^{R,f_{\mathsf{adapt}}}_{\mathsf{AdaptSig}}$.

This concludes the proof of Theorem 7. □

CHAPTER 7

Conclusion and Directions for Future Research

7.1 Conclusion

In this thesis, we provided foundations for the security of adaptor signatures (AS) as well as the applications using them as building blocks for blockchain protocols.

First, we showed a construction of an AS from isogenies that is provably secure in the post-quantum setting, specifically in (quantum) random oracle model, which also allows to achieve the security and privacy notions of interest for off-chain applications built upon it.

Next, we provided composable treatment of standalone cryptographic conditions as well as adaptor signatures within the Universally Composable (UC) framework. Towards this end, we gave novel ideal functionalities and defined concrete protocols which securely realize these functionalities.

We then defined a framework, dubbed LedgerLocks, for the secure design of AS-based blockchain applications in the presence of a realistic blockchain. For this reason, LedgerLocks is composed of a lock-enabling ledger that models AS-locked transactions and operates over the previously modularized cryptographic conditions and adaptor signatures.

Finally, we showcased the utility of our framework by using it to describe a payment channel and payment channel hub (PCH) protocols in a clear and modular fashion. The same blueprint can be used in the future to define other blockchain protocols based on AS-locked transactions.

7.2 Directions for Future Research

Throughout this thesis we focused on (two-party) AS built on top of (plain) digital signatures. However, in some settings we might need an AS scheme that provides additional capabilities than plain AS. For example, Qin et al. [QPM$^+$23] defined the notion of *blind adaptor signatures* and used it to construct a privacy-preserving PCH that support variable payment amounts. However, Qin et al. [QPM$^+$23] considered a fairly weak form of blindness[10] and provided only ECDSA-based construction. This brings us to the following natural question:

Question 1. *Can we generically construct (strongly) blind adaptor signatures?*

Blind adaptor signatures constitute only one potential extension of adaptor signatures to more advanced signatures. One can obviously also ask if we can construct adaptor signatures from other types of advanced signature primitives, such as ring signatures used in Monero or group signatures.

Question 2. *Can we (generically) construct adaptor signatures from other types of advanced signature primitives?*

Even though most cryptocurrencies base their transaction authorization on signature schemes shown to support adaptor signatures, there are some cryptocurrencies that do not utilize signatures for this. One notable example here is Zerocash [BCG$^+$14], which makes use of zero-knowledge proofs. Hence, it would be potentially interesting to come up with an adaptation functionality for zero-knowledge proofs, in order to embed some cryptographic condition within the proofs. This brings us to the following question:

Question 3. *Can we define and construct adaptor NIZK proof system?*

Given that there are cryptocurrencies that base their transaction authorization on other types of cryptographic primitives that do not support adaptor signatures, one can ask if we can extend LedgerLocks to also support these other types of ledger.

Question 4. *Can we extend LedgerLocks to account for other forms of transaction authorization?*

[10] In the *weak blindness* definition of Qin et al. [QPM$^+$23] the adversary should not be able to recover the message from a transcript but is allowed to link message/signature pair to a particular execution.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ADMP20] Navid Alamati, Luca De Feo, Hart Montgomery, and Sikhar Patranabis. Cryptographic group actions and applications. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 411–439, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.
→ Cited on page 2.

[AEE^{+}21] Lukas Aumayr, Oguzhan 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 Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part II*, volume 13091 of *Lecture Notes in Computer Science*, pages 635–664, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.
→ Cited on pages 1, 3, 6, 33, 34, 76, 85, 88, 90, 91, and 92.

[AME^{+}21] Lukas Aumayr, Matteo Maffei, Oguzhan 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*, pages 901–918, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
→ Cited on pages 3 and 76.

[AMKM21a] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Blitz: Secure multi-hop payments without two-phase commits. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 4043–4060. USENIX Association, August 11–13, 2021.
→ Cited on page 3.

[AMKM21b] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. Donner: UTXO-based virtual channels across multiple hops. Cryptology ePrint Archive, Report 2021/855, 2021. https://eprint.iacr.org/2021/855.
→ Cited on pages 3 and 76.

[ATM+22] Lukas Aumayr, Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. Sleepy channels: Bi-directional payment channels without watchtowers. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 179–192, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.
→ Cited on pages 3 and 76.

[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 *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.
→ Cited on pages 82 and 120.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 1–30, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.
→ Cited on pages 23, 25, 26, 27, and 121.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 103–112, Chicago, IL, USA, May 2–4, 1988. ACM Press.
→ Cited on page 9.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
→ Cited on pages 6 and 82.

[BHZ21] Christian Badertscher, Julia Hesse, and Vassilis Zikas. On the (ir)replaceability of global setups, or how (not) to use a global ledger. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 626–657, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.
→ Cited on page 75.

[BKV19] Ward Beullens, Thorsten Kleinjung, and Frederik Vercauteren. CSI-FiSh: Efficient isogeny based signatures through class group computations. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part I*, volume 11921 of *Lecture Notes in Computer Science*, pages 227–247, Kobe, Japan, December 8–12, 2019. Springer, Heidelberg, Germany.
→ Cited on pages 5, 8, 31, 32, 38, 54, and 55.

[BM22] Sergiu Bursuc and Sjouke Mauw. Contingent payments from two-party signing and verification for abelian groups. Cryptology ePrint Archive, Report 2022/719, 2022. https://eprint.iacr.org/2022/719.
→ Cited on page 3.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.
→ Cited on pages 6, 27, 75, 76, 78, 80, 82, 83, 84, 86, 88, 91, and 121.

[BS20] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 493–522, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.
→ Cited on page 9.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5), sep 2020.
→ Cited on pages 3, 5, 23, 24, and 25.

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 423–447, Lyon, France, April 23–27, 2023. Springer, Heidelberg, Germany.
→ Cited on page 9.

[CS20] Daniele Cozzo and Nigel P. Smart. Sashimi: Cutting up CSI-FiSh secret keys to produce an actively secure distributed signing protocol. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 169–186, Paris, France, April 15–17, 2020. Springer, Heidelberg, Germany.
→ Cited on pages 32 and 33.

[Dam99] Ivan Damgård. *Commitment Schemes and Zero-Knowledge Protocols*, pages 63–86. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
→ Cited on page 21.

[DEF18] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. FairSwap: How to fairly exchange digital goods. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 967–984, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
→ Cited on pages 75 and 76.

[DEFM19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*, pages 106–123, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.
→ Cited on pages 75 and 76.

[DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 949–966, Toronto, ON, Canada, October 15–19, 2018. ACM Press.
→ Cited on pages 75 and 76.

[DFMS19] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir transformation in the quantum random-oracle model. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 356–383, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.
→ Cited on pages 32 and 38.

[DG19] Luca De Feo and Steven D. Galbraith. SeaSign: Compact isogeny signatures from class group actions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 759–789, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.
→ Cited on page 8.

[DKL$^+$20] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.
→ Cited on page 36.

[DLL+17] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices. Cryptology ePrint Archive, Report 2017/633, 2017. https://eprint.iacr.org/2017/633.
→ Cited on page 56.

[DLRW23] Pierrick Dartois, Antonin Leroux, Damien Robert, and Benjamin Wesolowski. Sqisignhd: New dimensions in cryptography. Cryptology ePrint Archive, Paper 2023/436, 2023. https://eprint.iacr.org/2023/436.
→ Cited on page 36.

[DOY22] Wei Dai, Tatsuaki Okamoto, and Go Yamamoto. Stronger security and generic constructions for adaptor signatures. Cryptology ePrint Archive, Report 2022/1687, 2022. https://eprint.iacr.org/2022/1687.
→ Cited on page 65.

[EEE20] Muhammed F. Esgin, Oguzhan 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 2020: 25th European Symposium on Research in Computer Security, Part II*, volume 12309 of *Lecture Notes in Computer Science*, pages 378–397, Guildford, UK, September 14–18, 2020. Springer, Heidelberg, Germany.
→ Cited on pages 2, 3, 5, 20, and 56.

[EFH+21] Andreas Erwig, Sebastian Faust, Kristina Hostáková, Monosij Maitra, and Siavash Riahi. Two-party adaptor signatures from identification schemes. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12710 of *Lecture Notes in Computer Science*, pages 451–480, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.
→ Cited on pages 1, 2, 5, 12, 13, 17, 19, 20, 36, 63, 64, and 65.

[EMM19] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 801–815, London, UK, November 11–15, 2019. ACM Press.
→ Cited on pages 75 and 76.

[ER22] Andreas Erwig and Siavash Riahi. Deterministic wallets for adaptor signatures. In *European Symposium on Research in Computer Security*, pages 487–506. Springer, 2022.
→ Cited on page 3.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.
→ Cited on pages 2 and 10.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.
→ Cited on page 26.

[GMM$^+$22] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. Foundations of coin mixing services. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1259–1273, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.
→ Cited on pages xii, xiii, 2, 3, 76, 101, 102, and 122.

[GQ88] Louis C. Guillou and Jean-Jacques Quisquater. Efficient digital public-key signature with shadow (abstract). In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, page 223, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany.
→ Cited on page 20.

[GQ90] Louis C. Guillou and Jean-Jacques Quisquater. A "paradoxical" indentity-based signature scheme resulting from zero-knowledge. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 216–231, Santa Barbara, CA, USA, August 21–25, 1990. Springer, Heidelberg, Germany.
→ Cited on page 63.

[Gt19] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.2 edition, 2019.
→ Cited on page 54.

[HAB$^+$17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An untrusted bitcoin-compatible anonymous payment hub. In *ISOC Network and Distributed System Security Symposium – NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society.
→ Cited on pages 101, 102, 105, and 122.

[HK07] Omer Horvitz and Jonathan Katz. Universally-composable two-party computation in two rounds. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 111–129, Santa Barbara, CA, USA, August 19–23, 2007. Springer, Heidelberg, Germany.
→ Cited on page 107.

[HKL19] Eduard Hauck, Eike Kiltz, and Julian Loss. A modular treatment of blind signatures from identification schemes. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 345–375, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.
→ Cited on page 12.

[Hos21] Kristina Hostáková. *Foundations of Generalized State Channel Networks*. PhD thesis, Technische Universität, Darmstadt, 2021.
→ Cited on page 23.

[KL20] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. In Limin Jia and Ralf Küsters, editors, *CSF 2020: IEEE 33rd Computer Security Foundations Symposium*, pages 334–349, Boston, MA, USA, June 22–26, 2020. IEEE Computer Society Press.
→ Cited on page 75.

[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, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 33–61, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.
→ Cited on page 10.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498, Tokyo, Japan, March 3–6, 2013. Springer, Heidelberg, Germany.
→ Cited on pages 26, 27, and 121.

[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, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.
→ Cited on pages 27 and 64.

[Kup05] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35(1):170–188, July 2005.
→ Cited on page 9.

[Kup13] Greg Kuperberg. Another subexponential-time quantum algorithm for the dihedral hidden subgroup problem. In *TQC 2013*, pages 20–34, 2013.
→ Cited on page 9.

[KW03] Jonathan Katz and Nan Wang. Efficiency improvements for signature schemes with tight security reductions. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *ACM CCS 2003: 10th Conference on Computer and Communications Security*, pages 155–164, Washington, DC, USA, October 27–30, 2003. ACM Press.
→ Cited on pages 20 and 63.

[LGKK22] Thibaut Le Guilly, Nadav Kohen, and Ichiro Kuwahara. Bitcoin oracle contracts: Discreet log contracts in practice. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–8, 2022.
→ Cited on page 3.

[LNP22] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: Shorter, simpler, and more general. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 71–101, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.
→ Cited on pages 2 and 56.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616, Tokyo, Japan, December 6–10, 2009. Springer, Heidelberg, Germany.
→ Cited on page 2.

[Mir22] Arash Mirzaei. Daric: A storage efficient payment channel with penalization mechanism. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, pages 51–52, 2022.
→ Cited on page 3.

[MMS$^+$19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24–27, 2019. The

Internet Society.
→ Cited on pages 2, 3, 14, 74, and 90.

[MTV$^+$22] Varun Madathil, Sri AravindaKrishnan Thyagarajan, Dimitrios Vasilopoulos, Lloyd Fournier, Giulio Malavolta, and Pedro Moreno-Sanchez. Practical decentralized oracle contracts for cryptocurrencies. Cryptology ePrint Archive, Report 2022/499, 2022. https://eprint.iacr.org/2022/499.
→ Cited on pages 2 and 3.

[Pei20] Chris Peikert. He gives C-sieves on the CSIDH. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 463–492, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.
→ Cited on page 9.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Topics in Cryptology – CT-RSA 2016*, volume 9610 of *Lecture Notes in Computer Science*, pages 111–126, San Francisco, CA, USA, February 29 – March 4, 2016. Springer, Heidelberg, Germany.
→ Cited on pages 12 and 106.

[PS18] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel P. Smart, editor, *Topics in Cryptology – CT-RSA 2018*, volume 10808 of *Lecture Notes in Computer Science*, pages 319–338, San Francisco, CA, USA, April 16–20, 2018. Springer, Heidelberg, Germany.
→ Cited on page 12.

[QPM$^+$23] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oguzhan Ersoy, Amin Sakzad, Muhammed F. Esgin, Joseph K. Liu, Jiangshan Yu, and Tsz Hon Yuen. BlindHub: Bitcoin-compatible privacy-preserving payment channel hubs supporting variable amounts. In *2023 IEEE Symposium on Security and Privacy*, pages 2462–2480, San Francisco, CA, USA, May 21–25, 2023. IEEE Computer Society Press.
→ Cited on pages 2, 3, 76, and 120.

[Rob19] D. Robinson. HTLCs considered harmful, 2019. https://cbr.stanford.edu/sbc19/.
→ Cited on page 103.

[Rob23] Damien Robert. Breaking SIDH in polynomial time. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part V*, volume 14008 of *Lecture Notes in Computer Science*, pages 472–503, Lyon, France, April 23–27, 2023. Springer, Heidelberg, Germany.
→ Cited on page 9.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, January 1991.
→ Cited on pages 20 and 63.

[TM21] Sri Aravinda Krishnan Thyagarajan and Giulio Malavolta. Lockable signatures for blockchains: Scriptless scripts for all signatures. In *2021 IEEE Symposium on Security and Privacy*, pages 937–954, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
→ Cited on page 76.

[TMM21a] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. A^2L: Anonymous atomic locks for scalability in payment channel hubs. In *2021 IEEE Symposium on Security and Privacy*, pages 1834–1851, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.
→ Cited on pages xii, xiii, 2, 3, 76, 90, 101, 102, and 122.

[TMM21b] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. Post-quantum adaptor signature for privacy-preserving off-chain payments. In Nikita Borisov and Claudia Díaz, editors, *FC 2021: 25th International Conference on Financial Cryptography and Data Security, Part II*, volume 12675 of *Lecture Notes in Computer Science*, pages 131–150, Virtual Event, March 1–5, 2021. Springer, Heidelberg, Germany.
→ Cited on pages xii and 20.

[TMM22] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, and Pedro Moreno-Sanchez. Universal atomic swaps: Secure exchange of coins across all blockchains. In *2022 IEEE Symposium on Security and Privacy*, pages 1299–1316, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.
→ Cited on pages 1, 3, 75, and 76.

[TMSS20] Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmidt, and Dominique Schröder. PayMo: Payment channels for monero. Cryptology ePrint Archive, Report 2020/1441, 2020. https://eprint.iacr.org/2020/1441.
→ Cited on page 76.

[TMSS23] Erkan Tairi, Pedro Moreno-Sanchez, and Clara Schneidewind. Ledgerlocks: A security framework for blockchain protocols based on adaptor signatures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 859–873, New York, NY, USA, 2023. Association for Computing Machinery.
→ Cited on page xii.