Informatics

Formalization of Bitcoin off-chain protocols in F*

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

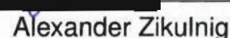
Alexander Zikulnig, BSc.
Matrikelnummer 11809909

an der Fakultät für Informatik

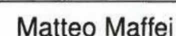
der Technischen Universität Wien

Betreuung: Univ. Prof. Matteo Maffei
Mitwirkung: Simon Jeanteur, MSc.
Magdalena Solitro, MSc.

Wien, 6. Mai 2024



Alexander Zikulnig



Matteo Maffei

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

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

TU WIEN Informatics

Formalization of Bitcoin off-chain protocols in F*

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

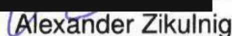
Alexander Zikulnig, BSc.
Registration Number 11809909

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Matteo Maffei
Assistance: Simon Jeanteur, MSc.
Magdalena Solitro, MSc.

Vienna, 6th May, 2024


Alexander Zikulnig

Matteo Maffei

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Erklärung zur Verfassung der Arbeit

Alexander Zikulnig, BSc.

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

Wien, 6. Mai 2024

Alexander Zikulnig

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Zuerst möchte ich meinen Eltern meinen Dank aussprechen. Sie haben mich immer tatkräftig unterstützt, sei es durch Ratschläge oder finanzielle Hilfe. Es ist immer wieder entschleunigend, sie in Kärnten zu besuchen, und eine große Freude, gemeinsam Zeit verbringen zu können. Ich bin ihnen zutiefst dankbar dafür, dass sie mir meine Ausbildung und auch das Leben in Wien ermöglicht haben.

Ein besonderer Dank geht an meine Freundin Annika, die mit ihrer aufmerksamen Art einen bedeutenden Beitrag zu meinem Studienerfolg geleistet hat. Sie war in den letzten Jahren meine emotionale Stütze und konnte mich immer wieder motivieren, weiterzumachen. Ohne sie wäre ich mit Sicherheit nicht die Person, die ich heute bin.

Ein spezieller Dank gilt meinen Mitstudenten und Freunden Julian Müllner, Andreas Lukitsch, Stefan Walser, Alexander Wieser und Michael Helm. Auch wenn der Kontakt mit einigen nach dem gemeinsamen Bachelorstudium etwas abnahm, erinnere ich mich noch immer an spannende Diskussionen, sowohl im als auch außerhalb des Hörsaals. Ich wünsche ihnen allen viel Erfolg beim Abschließen des Studiums und im weiteren Verlauf ihres Lebens.

Abschließend möchte ich mich auch bei meinen beiden Co-Betreuern Simon Jeanteur und Magdalena Solitro bedanken. Sie haben immer schnell auf meine Fragen reagiert und mit mir über neue Methoden diskutiert. Außerdem gaben sie mir wertvolles Feedback zu meiner Arbeit, das dringend notwendig war, um sie auf einen abgabetauglichen Stand zu bringen.

Es gab Momente, in denen ich meine bisherige Arbeit komplett umstrukturieren wollte, weil ich nicht zufrieden war. Zum Glück wurde mir dies von Simon und Magdalena ausgeredet. Ich habe gelernt, dass Perfektion ein langwieriger Prozess ist, der mehrere Iterationen benötigt und nicht immer innerhalb eines gewünschten Zeitraums umsetzbar ist. Um mit einem Zitat von Simon abzuschließen:
"Let's make something awful (now) so we know how to make something beautiful later on!"

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First, I would like to express my gratitude to my parents. They have always supported me, whether through advice or financial assistance. Visiting them in Carinthia is always a good way to relieve some stress and it brings me great joy to spend time together with them. I am deeply thankful to them for enabling my education and life in Vienna.

I want to thank my girlfriend Annika, who has made a significant contribution to my academic success through her attentive nature. Over the past years, she has been my emotional support and has consistently motivated me to keep going. Without her, I would certainly not be the person I am today.

A special thanks goes to my fellow students and friends Julian Müllner, Andreas Lukitsch, Stefan Walser, Alexander Wieser, and Michael Helm. Although contact with some of them decreased after our shared bachelor's studies, I still remember engaging in interesting discussions, in and out of the lecture hall. I wish them all success in completing their studies and in their future endeavors.

Lastly, I want to acknowledge my two co-supervisors, Simon Jeanteur and Magdalena Solitro. They have always responded promptly to my inquiries and engaged in discussions about new methods. Additionally, they provided valuable feedback on my work, which was essential to get it to a submission-ready state.

There were moments when I wanted to completely restructure all of my work done so far because I was not satisfied with it. Fortunately, Simon and Magdalena talked me out of it. I have learned that perfection is a lengthy process that requires multiple iterations and is not always achievable within a desired timeframe. To conclude with a quote from Simon:
"Let's make something awful (now) so we know how to make something beautiful later on!"

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Anzahl der Blockchain-Anwendungen ist in den letzten 16 Jahren enorm gewachsen und hat viele neue Forschungsbereiche geschaffen, die sich der Weiterentwicklung und Analyse dieser Technologie widmen. Eine der bekanntesten Implementierungen ist das Bitcoin-Netzwerk, das eine dezentralisierte Zahlungseinrichtung auf der Grundlage eines öffentlich zugänglichen Blockchain-*Ledgers* etabliert hat.

Trotz seiner vielversprechenden Eigenschaften hat seine wachsende Popularität auch dazu geführt, viele Probleme und Einschränkungen des ursprünglichen Designs aufzudecken. Ein spezifisches Problem, mit dem Bitcoin konfrontiert ist, während es versucht, sich im Bereich der *Decentralized Finance* zu etablieren, ist seine Skalierbarkeit, um einen höheren Transaktionsdurchsatz zu bewältigen. *Off-Chain*-Lösungen wie das *Lightning Network* haben den Ansatz verfolgt, die Blockchain zu entlasten und direkte Zahlungskanäle zwischen zwei Teilnehmern aufzubauen (daher *Off-Chain*). Mittels eines Konsensprotokolls tauschen die Parteien gültige *On-Chain*-Transaktionen aus, um ihren gemeinsamen finanziellen Stand zu aktualisieren. Dies kann unbestimmt oft durchgeführt werden, wobei das endgültige Ergebnis in der Blockchain eingetragen wird.

Solche Implementierungen stützen sich oft auf Timeout-Perioden, deren Einhaltung durch das Bitcoin-Netzwerk und seine Skriptsprache gewährleistet ist, sowie auf die Offenlegung geheimer Daten, um veraltete Zustände aufzuheben oder als Nachweis für erfolgreiche Zahlungen zu dienen.

Mit steigender Komplexität solcher Protokolle können manuelle Beweise über deren Richtigkeit zu fehlerhaften Ergebnissen führen. In dieser Arbeit legen wir den Grundstein für eine teilautomatisierte *Off-Chain*-Protokollverifikation in Bitcoin unter Verwendung der Programmiersprache F*. Wir erweitern das symbolische Verifikationsframework DY* um ein Blockchain-Modell, das es ermöglicht zeitliche Garantien zu überprüfen und einen neuen Label-Typ, um geheime Werte zu kennzeichnen, die gewollt offengelegt werden sollen, ohne dabei bestehende Sicherheitsgarantien (z.B., Geheimhaltung) zu verletzen.

Um die Richtigkeit des Blockchain-Systems zu überprüfen, werden eine Reihe von Lemmata, die Eigenschaften einer gültigen Blockchain beschreiben, als korrekt bewiesen. Zusätzlich stellen wir zwei einfache Protokollimplementierungen bereit, um die neu hinzugefügten Funktionalitäten vorzustellen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

The number of blockchain applications has grown immensely over the last decade, creating many new research areas dedicated to advancing and analyzing this technology. One of the most well-known implementations is the Bitcoin network, which established a decentralized payment facility, based on a publicly accessible blockchain ledger.

Despite its promising features, the growth in its popularity has also led to uncovering many issues and limitations of the original design. One specific problem Bitcoin encounters as it seeks to establish itself in the *Decentralized Finance* sector is its ability to scale as the transaction throughput grows. *Off-chain* solutions, such as the *Lightning Network*, aim to shift the load away from the blockchain (hence *off-chain*) and create direct *Payment Channels* between principals. In this way, the parties can publish on-chain only the transaction that establishes a new channel and the one that closes the communication, while all the intermediate exchanges are handled off-chain. Through a consensus protocol, participants exchange valid on-chain transactions to update their common financial state at their discretion. Once completed, the final result is recorded on the blockchain. Such implementations often rely on timeout periods, which are enforced through the Bitcoin network and its scripting language, and the disclosure of secret data to either revoke outdated states or as proof for successful payments.

As the complexity of such protocols increases, relying solely on manual reasoning for their correctness is insufficient, as it can be extremely time-consuming and error-prone. Hence, within this work, we provide the basis for a semi-automated off-chain protocol verification in Bitcoin using the proof-oriented programming language F*. We extend the symbolic verification framework DY* with a blockchain model supporting reasoning over time and introduce important modifications in the type system, allowing us to annotate secret values that will be intentionally disclosed without violating existing secrecy guarantees.

To verify the correctness of the blockchain system, a set of lemmas expressing properties of a valid blockchain are proven correct. Additionally, two simple protocol implementations are provided to introduce the usage of newly added functionalities.

TU Bibliothek, WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Problem Statement and Goal . . . 7
1.3 Contributions . . . 8
1.4 Outline . . . 9

2 Background **11**
2.1 State of the Art . . . 11
2.2 Bitcoin Blockchain . . . 16
2.3 The *Lightning Network* . . . 22

3 Type System of DY* **31**
3.1 Global Trace . . . 32
3.2 Labeling System and *Can Flow* Relation . . . 34
3.3 Valid Trace . . . 37
3.4 The `CRPYTO` and `LCRYPTO` Effects . . . 38
3.5 Witnesses and stable Predicates . . . 40
3.6 A dynamic Releasable Label . . . 43

4 Blockchain Formalization in F* **47**
4.1 Abstractions and Assumptions . . . 47
4.2 Blockchain Model . . . 48
4.3 `SCRIPT` Formalization . . . 56
4.4 Blockchain Protocol Verification . . . 63

5 Improvements and Future Work **67**

6 Conclusion **69**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures 71

Listings 73

Bibliography 75

Appendix 83

Blockchain API 83

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

The introduction of Bitcoin [Nak09] in 2008 has manifested the use of blockchain technology as a building block for decentralized applications and inspired an entire research field [GMAA22]. The core protocol of Bitcoin is based on a peer-to-peer network where transactions are recorded by the community on a public blockchain ledger, allocating coins to participating nodes by the *Unspent Transaction Output* (UTXO) principle. Protocols designed for Bitcoin (e.g., the Bitcoin core layer or protocols on top of it) often rely on smart contracts, to provide specific guarantees to honest participants when interacting with untrusted peers, together with the heavy use of cryptographic primitives. In the core layer, for example, public and private key signatures are used to create transactions and claim ownership of coins, and hash functions serve as tools to prevent the tampering or disclosure of secret values on the public blockchain.
These measures are necessary to achieve the main idea behind Bitcoin, namely the absence of a central trusted authority, managing the transfer and storage of funds.

Over the last 10 years, the trading price of Bitcoins has changed significantly: the course went from about 130$ in 2013 to nearly 60,000$ in November 2021, back down to about 16,000$ in December 2022, and hit a new peak in March 2024 of over 65,000$ [Sta23]. Despite this rapid growth of interest and willingness of people to buy Bitcoins, the underlying blockchain network suffers severe scalability problems [CDE+16, PD16, MMSH16] regarding transaction throughput, lacking far behind the performance of centralized financial institutions.

Off-chain protocols are one way to increase the number of transactions per second, by allowing several micro-transactions to be conducted directly between peers, without the need to write all of them to the blockchain [GMSR+19, MMSH16]. Those protocols are often called Layer 2 solutions, as they are developed on top of the core network layer, Layer 1 [GMSR+19].
Payment Channels (PC) are one example of off-chain protocols, enabling two parties to distribute funds, allocated inside a shared budget, between them. By allowing to

route transactions through multiple PCs (and hence via multiple peers), one can create *Payment Channel Networks* (PCN) [MMSH16, MMSK+17, GMSR+19], spanning yet another network layer on top of the peer-to-peer network. One of the most prominent examples of PCNs is the Bitcoin *Lightning Network* (LN)[PD16], developed in 2016.

The implementation of the LN utilizes *Hashed Timelock Contracts* (HTLC) [PD16] to securely transfer funds from one peer to another, via an arbitrary number of intermediate nodes. However, as it often happens, complex protocols hide subtle bugs that can lead to security vulnerabilities, and the LN is no exception: for example, three years after its creation, the *Wormhole* attack [MMSS+19] was found, affecting not only the Bitcoin LN, but essentially every construction built on HTLCs (with only two rounds of communication). As the main objective of the Bitcoin layer is to secure digital assets and transfer currency between individuals, security and the absence of logical faults in the protocol design are crucial. However, the combination of advanced cryptographic methods and the decentralization on top of a publicly accessible blockchain ledger make manual reasoning about protocols tedious and error-prone. Hence, a framework for the verification of off-chain protocols appears to be necessary.

Current research results in the area of formal methods in blockchain systems include statistical evaluation of attack probabilities (regarding inconsistent blockchain states) [CFvdPS15, LV20], verification of specific protocols [BGW20, MSMH21, KL20, GH22, GH23, Maz22], analysis of smart contracts [BZ19] through symbolic execution [JD23, KB18, BBKM23], and formal modeling of blockchain and ledger-based systems [KK18, ABLZ18]. However, besides some of the recently published work mentioned above, there appears to be little literature tackling the formal analysis of protocols implemented on top of the Bitcoin core layer.
Some of the identified approaches rely on established formal tools such as Tamarin [MSCB13], ProVerif [Bla14], and the OFMC symbolic checker [BMV05], or deploy custom methods based on input translation together with logical resolution or theorem proving. Still, most methodologies require compact abstract protocol models (e.g., a specification in spi-calculus, CSP, or domain-specific rules) [BGW20, GH22, GH23], which may be far away from an actual protocol implementation in software, manual proof assistance (as problems get too big for the applied tool to handle it) or hand-written proofs [GH23, KL20], or simply do not scale well to accurately represent elaborate protocols (e.g., unbounded number of players or protocol loops) [BGW20, GH22, GH23].

Type systems, on the other hand, seem to be a scalable solution for verification of even complex protocols [BFG10]. Moreover, literature shows that elaborate type systems can efficiently be implemented in software [BHM14, BBD+21a, BBF+08, CGLM17].
Recent developments on the verification framework DY* [BBD+21a, BBD+21b] indicate promising results of automated verification using type systems. DY* is a symbolic verification framework, that supports trace-based reasoning in the presence of a *Dolev-Yao* attacker[DY83] and (in theory) executable protocol implementations. It is written in the proof-oriented programming language F* [SHK+16, Nik24], which supports a mix of fully automated and manually guided proofs discharged by the SMT solver z3 [dMB08],

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

dependent and refinement types, and user-defined computational effects. The DY⋆ base layer thus far does not include mechanisms to model off-chain protocols.

This thesis presents a formalization of a blockchain system in F⋆ to extend the DY⋆ framework, and thereby provide techniques to allow modeling of Bitcoin blockchain protocols. Eventually, further extension should lead to the verification of HTLCs and off-chain PCNs.

1.1 Motivation

Without rigorous verification, subtle errors can remain unnoticed for several years and lead to major security breaches in the long run. One well-known example that emphasizes the significance of verification is the Needham-Schroeder (NS) protocol [NS78]. The NS protocol was introduced in 1978 and was designed to established secure communications between two peers over an insecure network. It was considered secure for 17 years until Gavin Lowe discovered an attack in 1995 [Low95] by applying model-checking techniques. Today, blockchain technology represents the main building block of applications in the *Decentralized Finance* (DeFi) sector [DSST+22]. Unidentified bugs in DeFi systems are a main concern, as they can potentially cause serious financial damage. For this reason, a thorough security analysis is indispensable.

Scalability of Bitcoin

As aforementioned, because of its growing number of participants, Bitcoin (and also other cryptocurrencies built on the Bitcoin core) suffers scalability issues. The network can only process below 10 transactions per second, whereas the leading financial service institutions, such as Visa, can support up to several thousands [MDPM18, Vis23]. This bottleneck stems mainly from the decentralized consensus and *proof-of-work* concept, limiting the farming rate at which miners create new blocks, to an interval of approximately 10 minutes. On average, blocks mined in July 2023, contain between 1800 and 3800 transactions [Blo24], which equals a transaction rate of roughly 3 to 6 transactions per second.
Also, high processing fees, paid to miners as an incentive to favor certain transactions, make small payments via Bitcoin unattractive. At its peak, in April 2021, average fee prices were as high as 27$ [Sta22].
These problems primarily drive the research on new scalable solutions, such as increased block size (to hold more transactions), efficient consensus mechanisms [LNB+15], off-chain transactions, sharding [WSNH19] or novel blockchain implementations [DOA16].
The SegWit [Eri15] update (Segregated Witness) of 2017, almost doubled the effective number of transactions per block, by excluding signature from transaction data. Besides the slight increase in throughput, the adaption of SegWit enables the integration of off-chain PCs, as the computation of the transaction identifier is now independent of signature data (see Section 2.3 for details).

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

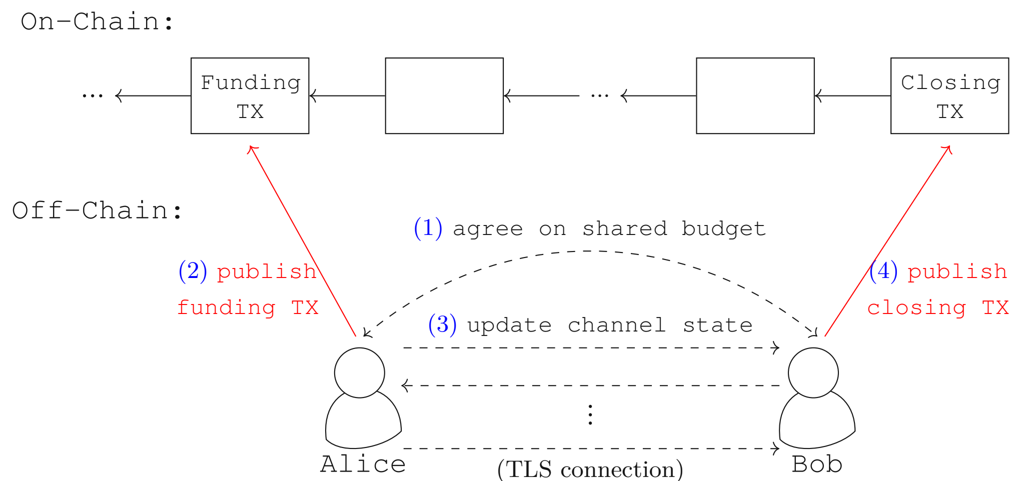


Figure 1.1: Illustration of a LN *Payment Channel*

Payment Channels

Classical Bitcoin transactions are executed *on-chain*, as they are broadcasted to the network and added to a block on the blockchain. Hence, anyone with access to the current version of the ledger can observe and verify the existence of a certain transaction. To spend the output of some transactions, one usually waits until it gets confirmed: this happens when a sufficient amount of new blocks (usually at least 6) is added after the block holding said transaction. The confirmation ensures that the transaction is part of the longest blockchain version and is thus seen as valid by the network. Confirmation of on-chain transactions can take a significant amount of time.

The channel construction defined by Poon et al. [PD16], interacts with the blockchain only to open or settle (long-term) payments and resolve conflicts that arise due to protocol violations. Thereby, the payment process is shifted away from the *slow* blockchain. The protocol is illustrated in Figure 1.1 In bidirectional payments, a shared budget is created through an initial funding transaction published on-chain, allocating coins from one or both channel participants. Then micro-transactions are committed by exchanging pre-signed on-chain transactions, that define a new distribution of funds from a shared budget. Each transaction establishes a new channel state and can be published on the blockchain anytime. By choosing not to publish them, the channel is kept open and further state updates are possible, but as soon as one is published, the channel is closed and the current payment is settled according to its defined state. State updates are exchanged via secure communication paths outside the blockchain (e.g., TLS). Micro-transactions are valid upon receipt, thereby eliminating long confirmation periods.
To invalidate outdated states, the parties exchange private signature keys, that allow the punishment of a dishonest principal who published an invalid state, by collecting all of their shared funds.
This punishment mechanism also depends on timelock features that are part of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bitcoin consensus protocol [Pet14, Mar15a], which defines absolute and relative timelocks for transaction validity.

PCNs (e.g., LN) extend simple PCs by connecting already existing channels and route between them, rather than opening new ones, to avoid additional fees and long confirmation periods. A payment that takes place over a PCN is called multi-hop payment, highlighting the fact that the payment "hops" through different nodes before arriving at its destination. To ensure the designated payment recipient actually receives the coins, PCs of the LN are augmented by HTLCs, which enforce a certain behavior of intermediate hops. HTLCs also rely on the disclosure of secret values, as proof for successful payments, and timelocks to issue refunds after a timeout period.

Further details about off-chain payments are presented and discussed in Section 2.3.

Security and Privacy

The blockchain ledger, per construction, promises certain guarantees, as non-malleability of transactions (once they are confirmed), pseudo-anonymity, and transparency (i.e., each peer can check the history of transactions themselves and verify them). Still, Nakamoto had already identified privacy issues, arising from analyzing chains of transactions and pooling together addresses, believed to be owned by one person or organization, as it is briefly described in the Bitcoin white paper [Nak09]. By applying clustering heuristics on tagged addresses, links to real-life identities are possible (depending on the available data) [MPJ+16]. Government authorities, for example, may force crypto-exchanges to reveal user information (in case of possible criminal activities), to match issued addresses with individuals.
Off-chain protocols could provide stronger notions of privacy if they are implemented correctly, as not every transaction is traceable via the blockchain. Still, poorly designed protocols may destroy such guarantees nonetheless, by revealing pieces of information on the network, that allow actions to be linked with concrete pseudo-addresses.
The work of Malavolta et al. [MMSK+17, MMSS+19] analyzes security and privacy issues in PCNs. They found privacy issues due to the leakage of common identifiers when using HTLCs.

HTLCs are contracts set up between two individuals to lock funds until conditions to distribute them are met. As an analogy, one can think of a payment process that includes shipping goods, where a trusted third party keeps the money of the payer locked up until the goods arrive, and afterwards gives the money to the payee. The locking mechanism depends on a hash and the unlocking of the corresponding preimage. Along a payment route in the LN, intermediate hops lock their coins with HTLCs but they all use the same hash. This enables observing intermediate nodes to identify the endpoints of the current transactions (i.e., who pays who) and in some cases even the entire payment route [MMSK+17]. *Multi-Hop HTLCs* (MHTLC) have been proposed to counter those issues, while still being compatible with the Bitcoin scripting language. The new contract requires different but related hash preimages on every intermediate HTLC [MMSK+17].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The communication effort is rather high, compared to standard LN payments: each hash has to be sent by the receiver of the payment to all other nodes on the payment route. In [MMSS+19] a more efficient and privacy-preserving lock construction is introduced, so-called *Anonymous Multi-Hop Locks* (AMHL), based on *Adaptor Signatures*.

To open new channels, an initial transaction containing the channel capacity is published (shared budget). The referenced coins are spendable by a multi-signature (also called *multisig*) address, meaning that both PC participants have to collaboratively provide a signature to move the funds. Currently, the share of output scripts that include pure multi-signatures is far below 1% [JD23] (not considering "script-hash" payments), hence such transactions are rather peculiar when seen on-chain.
The adaption of Schnorr signatures in the Bitcoin core, instead of the currently applied ECDSA signatures, could mitigate privacy concerns of mutlisig transactions [MPSW19]. The *Bitcoin Improvement Proposal* (BIP) [Pie20], discussing the implementation of Schnorr signatures in Bitcoin, lists multi-signature transactions as one potential application, where the key and signature aggregation properties of Schnorr pose an advantage over ECDSA. Multiple peers would be able to jointly sign a transaction by combining their individual signatures into one, and derive a single verification key, using their public key material. From a third-party perspective, there would then be no difference between multi-signature and single-signature transactions.

In [MMSK+17, MMSS+19] the authors formulate a set of security and privacy properties applicable in the PCN setting. Such formulations are important to evaluate the performance of protocols. The specification includes, among others, requirements concerning *Atomicity*, *Relationship Anonymity*, and *Balance Security*. Intuitively, these properties define, on a higher level, how payment operations should ideally behave from a user's point of view:

- *Atomicity*: Along a single payment route, if a PC changes its state (i.e., the channel capacity updates), then the same change is applied to all other PCs on that route. This means that all related HTLCs have the same outcome, and the corresponding payment either went through and all channel states were updated, or it failed and the states stayed the same.
- *Relationship Anonymity*: An intermediate node does not learn any information about other participating nodes in a payment process, except its direct neighbors.
- *Balance Security*; An honest user can not lose coins by participating in a payment process, even if all other participants exhibit malicious behavior.

Even though standard HTLCs may not provide all the security and privacy requirements, as defined by Malavolta et al. [MMSK+17, MMSS+19], they are as today still part of the LN specification [LN 16] and an important building block to construct networked channels. They are also deployed in other protocols, such as cross-chain swaps [Maz22]. Hence, the ability to easily model and verify HTLC-based protocols remains an essential part of improving the security of current blockchain applications.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.2 Problem Statement and Goal

Thorough formal analysis is essential to protocol construction, as it allows to identify logical flaws and provide proofs of specific security properties. While successful tools exist, such as ProVerif [Bla14] or Tamarin [MSCB13], protocols with an unbounded number of rounds or players remain often infeasible to analyze. LN or PCNs in general fall in that category, as the number of intermediate nodes taking part in a multihop payment is arbitrary. On the other hand, type systems shine with this kind of problem thanks to their strong compositional and inductive capabilities. For instance, the DY* framework [BBD+21b, BBD+21a] proposes an efficient type system based on semi-automated protocol verification.

The significance of off-chain applications using HTLCs, particularly PCNs [MMSH16, MMSK+17], as well as security and privacy concerns caused by faulty protocol implementations, has been well-established in the previous section. In light of this, we argue that for further advancement of blockchain technologies, analysis of off-chain PCs is of high importance. Hence, this thesis focuses on providing new ways to contribute to the verification of off-chain systems and eventually HTLCs, as they are implemented in the LN, and PCNs that incorporate them. Because of the aforementioned limitations of standard verification tools, this work relies on and extends the DY* framework.

In Section 1.1, we briefly review PCs and multi-hop payments (for more details see Chapter 2, especially Section 2.3.1 and Section 2.3.2) and emphasize that both constructions deliberately reveal secret data, either to invalidate outdated channel states (exchange of private keys) or as evidence that a payment has been successfully completed (preimage of hashes in HTLCs). Additionally, payment channel implementations impose timing constraints by restricting transaction validity with timelocks.

To successfully verify such protocols within DY*, we identify two main sub-goals that create a basis for the verification process:

1. Modify DY*'s type system to allow the intentional release of secret values to the network.

2. Formalize a model of the blockchain that supports time-dependent reasoning over block height.

(1): The property of a value x being secret until some event happens makes verification with the prevalent framework definition impossible, as secrecy is of major concern (i.e., no disclosure of secret values). The property is difficult to formalize with type-based proof assistants, as in contrast to non-type system tools, the notion of secrecy is defined through type labels.

DY* provides a labeling system to annotate data objects according to their security level. However, the system is not expressive enough for our cause, as it is static and labels of objects can not be switched once they are defined. Although DY* includes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

session state compromising features to reveal private data to the attacker, its semantics obviously differ from voluntary disclosure. For these reasons, we need to augment the labeling system to make it suitable for our purposes. More precisely, we defined a new type of label with the following property: when a certain entry appears in the trace, the label (which is initially private) can flow to public labels. Similar ideas to construct such a typing definition are found in the type system of [BHM14]. Here, the type `private` is defined as a function from `unit` to `Un` (the type of values known to the attacker), with refinement `assert(false)`. Hence, once the private value is disclosed (i.e., the function is executed), the assertion will fail, prohibiting data leakage. Tweaking the refinement, to `assert(C)`, where `C` is some (arbitrary) condition, would allow disclosure provided `C` holds.

(2): A blockchain model for off-chain transactions should support the reasoning over timeout periods. Fortunately, DY⋆ is built around a global trace, which serves as history protocol execution steps. Each index of the trace can be seen as a time instance and thereby an ordered relation between each entry is created. Bitcoin most commonly refers to timing constraints using *block height* or *depth*. Both variants can be expressed using just the list of block entries and their order. A related definition is given in [ABLZ18], which presents a stripped-down formal model of Bitcoin transactions, using a list of single transaction blocks.

We split up the work into those two parts mentioned above, with this thesis focusing on the blockchain part **(2)**. Thus, this thesis aims to contribute to the goal of automating the verification process of blockchain protocols and especially off-chain based ones, like those implemented in the Bitcoin *Lightning Network*, by extending the DY⋆ protocol verification framework with a blockchain model formalization.
Work on **(1)** is currently in progress, and eventually both pieces will be merged together. For the sake of completeness, current preliminary results are also briefly presented in Section 3.6.

1.3 Contributions

This thesis covers the following contributions and extensions to DY⋆:

- A Bitcoin blockchain model integrated into DY⋆, supporting the notion of time (block height).
- Formalization of the Bitcoin scripting language `SCRIPT` in F⋆.
- Predicates to reason about timelocks semantics of Bitcoin transactions, as needed for off-chain protocols.
- User API functions to ease blockchain protocol implementation.
- An use-case example of a simple protocol that redeems coins on the blockchain.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- A preview of a new label type to intentionally disclose secrets

The implementation includes an interpreter for Bitcoin scripts, including a selection of opcodes listed on the Bitcoin wiki page [Bit24b]. Scripts can be written within the framework as a list of opcodes. The parsing of real-world binary scripts is currently not supported, due to the lack of available cryptographic functions in F*. The interpreter can be used to either simply evaluate scripts through normalization or create (limited) manual or semi-automated proofs about script execution.

The blockchain model supports time-depend protocols by reasoning over block height (or depth) as implemented in the Bitcoin blockchain (see Section 2.2.3 for a deeper dive). Furthermore, our new label type allows for the verification of protocols with intentionally leaked secrets. Having both functionalities in place paves the way to verifying off-chain related protocols, like HTLCs and the LN (see Section 2.3 for details on secret disclosure and timeouts in PCs).

The project also includes simple examples using the newly integrated functionality. We added a small protocol used to redeem coins on-chain to showcase the blockchain machinery. This demonstrates the necessary steps to perform verification and also shows the capabilities of added API functions.
Additionally, there is a simple test protocol for releasable labels included.

There has currently not been any attempt (at least known to us) of applying DY* to verify protocols in Bitcoin or within a blockchain setting.

1.4 Outline

After providing an introduction to off-chain protocols, mentioning scalability issues of Bitcoin, and discussing security and privacy concerns of PCs, Chapter 2 provides technical background information. Section 2.1 gives an overview of state-of-the-art verification methodologies, and highlights some current work applying formal methods to Bitcoin. In Section 2.2 important concepts of the Bitcoin blockchain are introduced, while Section 2.3 dives deeper into the workings of the *Lightning Network*.

Subsequent chapters focus on the extensions made during the work on this thesis: First, Chapter 3 gives a quick overview of DY*, encompassing its global trace, the labeling system, and invariant-based protocol reasoning. Chapter 4 builds on that knowledge, by defining the blockchain model and how it is integrated into DY*. Information regarding script parsing and time reasoning is also found here. Lastly, further improvements and possible future work are discussed in Chapter 5, and finally, Chapter 6 concludes this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Background

2.1 State of the Art

In the previous chapter, we discussed the need for automated security proofs and highlighted problems of current blockchain protocols. This section first gives an overview of current verification approaches and available tools and introduces the DY* framework. Then we show some of the recent research results in the area of formal verification targeting Bitcoin.

2.1.1 Verification Methodologies and Tools

Formal methods in security is a vast research field, hence many different ways of modeling and reasoning about protocols exist. We can, however, identify two main branches to categorize tools, namely the *symbolic model* and the *computational model* [Bla12].

Computational Model. The computational model is mostly common among cryptographers, who mathematically reason about cryptographic schemes (using game- or simulation-based proofs). In this model, messages are represented as strings of bytes, the adversary is a polynomial-time Turing machine, and cryptographic primitives are expressed through mathematical equations: the goal is to show that the adversary has at most a negligible probability of breaking the scheme [Bla12]. The computational model works with a detailed and rigorous model of the cryptographic primitives; therefore, while the resulting proofs can ensure a high level of accuracy, they require significantly more effort to be produced, making the computational approach not scalable for large and complex protocols.

Different verification tools based on the computational model were engineered over the years. One of the most successful is CryptoVerif [Bla06], a computationally sound

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

automated protocol verifier with support for secrecy, correspondence, and (some) equivalence properties on stateless protocols. It functions by emulated game-based proofs typically done by hand by a cryptographer.
Another popular tool is Squirrel [BDJ+21], based on the Bana-Comon logic [BCL12], which brings computational soundness to a more symbolic style of proofs with a focus on observational equivalence.

Symbolic Model. When the protocols we deal with are too large and complex to verify with the computational model, we can rely on a symbolic approach. This method treats cryptographic functions as reliable black boxes: it reasons over the protocol taking into consideration only the extensional properties of such functions (i.e. what inputs they take and what outputs are provided in return), while the details of the implementation are blurred out [Bla12]. This places the focus on aspects of the underlying protocol, rather than the cryptography involved. Furthermore, the higher level of abstraction makes automated verification more manageable than in the computational case.

One of the most prominent tools within the symbolic world is ProVerif [Bla14], which translates the input model (in applied pi-calculus) to a Horn-clause representation and applies resolution over it against a logical model of the attacker (*Dolev-Yao* model [DY83]). ProVerif can verify several security properties in a fully automated way, such as secrecy and authentication, and supports differential equivalence (strong equivalence for similar processes, differing only by their terms). Since the logical derivation resembles protocol execution traces, counterexamples of queries can be provided.
On top of ProVerif, the (executable) modeling language ProScript [KBB17] provides a higher level abstraction from process calculi. Models in applied pi-calculus are extracted directly from ProScript code (that can be run within JavaScript), which are used for further verification in ProVerif. The soundness of the translation to applied pi-calculus has yet to be proven and also the software implementation is currently in an experimental state.

The Tamarin [MSCB13] prover employs a constrained-solving algorithm [SMCB12], utilizing a set of logical rules and reduction steps that specify a transition system. Security properties (trace-based and observational equivalences [BDS15]) are written as first-order formulas and proofs can be guided through lemmas in an automated or interactive way. Similar to ProVerif, an unbounded number of sessions are supported and counterexamples are generated. Additionally, protocols with mutable-state can be verified. Tamarin has already been used to verify HTLC channels between two peers within a simplified blockchain model [BGW20].

Type Systems. The revised tools tackle the problem of protocol verification directly, by translating the whole input model into logical formulas and subsequently verifying them via automated reasoning. This comes with the cost of abstraction (a model has to be formulated) and scalability (e.g., state-space explosion through unbounded attacker, composite or unbounded protocols), as the analysis for elaborate protocols can take a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

significant amount of time [BFG10] (see verification of Signal protocol in ProVerif [KBB17]) and human effort regarding model design. To counter the abstraction problem, the authors of [BFGT06] proposed, back in 2006, to extract models for ProVerif directly from F# implementation code, following a similar idea as ProScript. They argue that, by constructing models from the ground up, solely based on the protocol specification, many bugs are left unrecognized, since they are most likely introduced during development.
Scalability problems can be solved by type systems [BFG10], as they naturally handle compositional reasoning, allowing modules to be checked independently of others (hence they can be verified once and for all) and support inductive reasoning of protocols with unbounded loops [BBD+21b]. Tools like ProVerif or Tamarin have no or only limited support for such operations. As an example consider, blockchain protocols, like multi-hop payment channels, which allow an arbitrary number of participants. In [BBF+08] Bengston et al. presented the F7 type checker for protocols written in F# and compared it against previous work in [BFGT06]. Their evaluation showed an improvement in terms of performance and the variety of protocols that can be verified (e.g., recursive functions), however, they also highlight the lack of support for certain properties, like equivalences.

DY⋆ Library. DY⋆ [BBD+21a, BBD+21b] is a symbolic verification framework, that integrates a security type system (similar to [FM11, BHM14]) on top of the programming language F⋆ [SHK+16]. The core of the library is built around a global trace, which tracks triggered events, messages sent on the network, and the state and ongoing sessions of principles within a single protocol execution. By reasoning over trace entries and their order, trace-based properties such as standard secrecy, authentication, and forward secrecy can be verified, but not equivalences. The trace includes entries referring to the state of principals, hence the class of stateful protocols is supported. Modeling stateful protocols is only limited possible for example with ProVerif, without further extensions [YXL+22]. The authors of the framework define a valid trace to adhere to a set of invariants (generic and protocol-specific). Requirements of protocols are captured via said invariants (e.g., *"When can certain events be triggered?"* or *"How can a certain state be reached?"*), and security properties are stated in the form of F⋆ lemmas, which are verified against the valid trace definition. A (*Dolev-Yao*) attacker can compromise any state entry within the global trace (even old ones), thus gaining access to secrets originating from that entry. Protocols are written in annotated F⋆ code and can be compiled to OCaml (using a concrete library for cryptographic operations) and from there to Assembly. Furthermore, by annotating the protocol code the designer is guided during the implementation process, to obey the specified (sound) typing rules.
The authors [BBD+21b] claim that verification with DY⋆ requires less effort than with comparable approaches, like F7. We give a detailed introduction to DY⋆ in Chapter 3.

2.1.2 Formal Methods in Bitcoin

There are different ways to show that a program is secure. One of these is leveraging equivalence properties to prove that the executions of two different programs are in-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

distinguishable from the outside (by an adversary), given their externally observable behavior. For instance, the *Universal Composability* (UC) framework [Can01] uses this approach to define and prove security properties in a modular and composable way, by comparing the protocol's real-world specification against an ideal functionality. Desired security properties are integrated into the design of the ideal functionality. By showing that the real-world implementation is indistinguishable from the ideal version, the former inherits all specified properties of the latter. If one can verify the security of the ideal functionality, the security of the real-world protocol follows.
Kiayias et al. [KL20] modeled the LN specification [LN 16] within the UC framework. They provided a proof-by-hand, which led them to the conclusion that honest principals running programs adhering to the LN specification do not lose funds due to the malicious behavior of peers.
Grundmann et al. [GH20, GH22, GH23] build upon the work of [KL20], with the goal of automating some parts of the manual proof using model checking tools and the temporal language TLA+. They show that their formalization refines the ideal abstraction, and the abstraction implies the desired security goal: malicious peers are not able to steal coins. The authors mention problems regarding large state spaces when modeling an entire payment network. In their work, Grundmann et al. dealt with this problem, by also refining PCs with an ideal functionality, limiting the number of possible channel states. The provided proof sketch considers a multi-hop system with three users and two PCs. During their work, they identified errors in the formalization of [KL20].

Atzei et al. define a formal model of Bitcoin transactions [ABLZ18], including multi-signatures, segregated witness data, and scripts. The model can be applied in combination with existing blockchain formalizations to enable reasoning about the Bitcoin blockchain and smart contracts. Scripts are not defined in a stack-based manner but are formulated as simple grammar over expressions together with semantics and evaluation procedures. The grammar allows simple arithmetics, conditional branching, signature verification, hashes, and two temporal operators, expressing absolute and relative timelocks. Using their definitions the authors demonstrate how simple requirements of the blockchain, such as the absence of double-spending transactions, can be expressed and evaluated in their model. Additionally, a compiler is provided, which can translate modeled transactions into real Bitcoin transactions.

The work in [BGW20] focuses on establishing a simplified formal blockchain model in Tamarin. The authors identified some key aspects the model should support to allow time-based protocol verification, such as the general notion of time and the ability to trigger timeout events. The base construct of the model is the so-called Tickchain, representing a linear increasing timeline. Each entry in the Tickchain has a corresponding ledger entry, containing zero or one transaction, thus, enabling a variable timing definition (e.g., block height or others). The model supports only two transaction formats: simple transactions, transferring coins from an input to a public key address, and commitment transactions, containing an HTLC output script instead of an address. To demonstrate their model the authors perform an analysis of an HTLC-based cross-chain swap protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(e.g., exchange Bitcoins for Altcoins) between two parties. Only one initiator and receiver were modeled, thereby ignoring payment channels with multiple hops. Analyzed security properties focus on temporal constraints, that limit the action space of participants until certain transactions have been created or events occurred (e.g., timelock expires). Their model is powerful enough to identify problems when establishing HTLCs across blockchains evolving at different rates, as timeout periods might not match anymore. The proof, so far, needs manual guidance.

Modesti et al. [MSMH21] analyzed the Bitcoin *Payment Protocol*, introduced in BIP70 [Gav13], to formally show the protocol's vulnerability to the *Silkroad Trader* attack. The protocol describes the interaction of a customer with a merchant to buy goods using Bitcoin. The authors formulate a model of the protocol in the `AnB` language and verify a single session with the OFMC model checker. Their verification goals include authentication and agreement over the issued refund addresses, by the customer to the merchant. Results show that the merchant is not able to verify that the customer agrees on the refund addresses, as the `Payment` message (which includes the refund addresses) is missing a signature linkable to the customer's payment. Malicious parties may take advantage of this bug, by moving coins to illicit traders, from whom they acquire goods, via an honest merchant, without being directly involved in the payment process.
Two fixes have been proposed and the specified authentication goals could be proven correct.

In [LV20] the authors established a probabilistic Bitcoin network model as a stochastic process of mining instances instantiated in parallel. During their work, they augment the probabilistic model checker PRISM with blockchain-related datatypes, which they call PRISM+. The enhanced model is used to compute the probabilities of attacks resulting in inconsistencies of the blockchain (e.g., double-spends). For the example provided in the paper, the model is used to calculate the probability of a malicious miner (or mining pool) creating an alternate longest version of the blockchain depending on its available hashing power and the depth of the fork where the miner starts.

The tool CHAUSETTE [JD23] performs a vulnerability analysis of non-standard Bitcoin scripts, via symbolic execution. The author's goal is to identify possible attacks to steal coins from not sufficiently secured transactions. They gathered data from the first 775.000 blocks of the Bitcoin blockchain and created a dataset of all the script types found. The data shows that 3.868.544 non-standard scripts could be pinned down, whereas 433.458 of them represent output scripts (i.e., are still spendable) and 3.435.086 are redeeming scripts (i.e., used to spend an output). The symbolic execution translates the scripts into a logical formula and queries z3 [dMB08] to find a solution for inputs which leads to the acceptance of the script (non-zero value on top of the stack). In total, they discovered more than 300.000 unsafe scripts using their method.

Klomp et al. [KB18] introduced a symbolic verification tool, to check for constraints on input scripts imposed by a given output script. The small-step evaluation is implemented in the functional programming language Haskell. When run on a script, the tool gathers constraints about the initial stack elements in all possible execution paths and queries a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Finite Domain Constraint Solver in the back-end. The output is a description of how the stack has to be set up by the input, such that the analyzed script executes successfully. In another paper, the authors continue with this work and present a framework for verification of decentralized applications in Bitcoin, based on the symbolic execution of open scripts, called SCRIFY [BBKM23]. They show how generated constraints over scripts can be used to reason about protocol behavior, using the example of HTLCs in the context of PCs.

Besides off-chain solutions, there exist other approaches to mitigate the scalability issues encountered by blockchain-based networks, such as Bitcoin. A *Temporal Blockchain* [DOA16] has been proposed aiming to keep the size of the blockchain ledger as small as possible. The authors argue that the Bitcoin blockchain is to large in terms of storage space (currently over 500GB [1]) for most people to consider running a full node and actively participate in the network, as they would have to download the entire ledger. In contrast, they claim that their implementation keeps a fixed size of only 4.5GB despite continuous network growth. This involves miners periodically creating checkpoints and removing blocks of a specific "age", to keep the size constant.
By recruiting more active nodes, the network not only becomes publicly more accessible but should also allow more resource-limited users to provide hashing and computing power to the network. Doing so, the authors see an improvement in Bitcoin's security, as they find 51%-attacks to get more unlikely and resource intensive.
They conduct a formal analysis through model checking by specifying their blockchain in the B-language and evaluate against an attacker with less than 50% of the available total computing power. Due to the lack of well-established blockchain security definitions at the time, the authors focused on key properties that blockchains should provide, such as the non-malleability of blocks (given blocks are removed regularly by miners). Their results claim the *Temporal Blockchain* provides the same security guarantees as the standard blockchain but needs to undergo further research to gain confidence the design.

2.2 Bitcoin Blockchain

The Bitcoin blockchain [Nak09] is a collection of *transactions* enabling coins to flow from one to the other given a set of conditions is satisfied (for details on the implementation of the Bitcoin core refer to [Bit24a]). The transactions are gathered in *blocks* over which a decentralized consensus algorithm is run (proof of work in Bitcoin's case). The consensus ensures the blockchain is kept in a valid state, without having a central authority in place.

The blockchain is often referred to as a public ledger, that contains the history of all transactions that have ever been recorded. It follows the *Unspent-Transaction-Output* (UTXO) principle, meaning that it ensures the consistency of each transaction (through consensus), but notably, does not track the balance of its users. An agent "owns" the

[1] https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/, accessed 6-November-2023

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

coins of any output from which they have enough information to claim. The ledger is maintained by having each block reference its predecessor block, thus creating the eponymous chain.

Bitcoin blocks can contain an arbitrary number of transactions (bounded by the maximum size of blocks), that consume coins from previous transactions and transfer ownership of said coins. Each transaction has a list of at least one[2] or more inputs and outputs (see Figure 2.1). The inputs define where to pull coins from, while the outputs describe *how* and how much of those coins can be redeemed. The output *locks* coins by defining a small program, written in a simple (non Turing-complete) stack-based scripting language, called `SCRIPT` [Bit24b]. An input references a specific transaction output on-chain and has to provide information to satisfy the output's script. The input also passes the required data in the form of a script. Evaluation is done by running the interpreter on the concatenation of both scripts, input and then output. Hence the input's script sets up the stack on which the output's script is run. Most commonly, output scripts require a signature associated with a public key of the payment recipient, but they can even contain more complex conditions (including timing restriction or the knowledge of some secret), or simply trivially satisfied programs (see [JD23] for some examples).

Transactions within a block are independent of one another. Blocks and transactions are "uniquely" identified by their block identifier (BID) and transaction identifier (TXID) respectively. The TXID is computed as the double hash of the serialized transaction [Eri15]. The BID is the hash of the block header [Bit21], which includes meta-data like the hash of the previous block, a nonce related to the *proof-of-work* computations, and a merkle-root hash of all the block's transactions. The merkle-root commits a block to its list of transactions. Hence, changing the contents of one transaction will have effects on the respective TXID and BID. Transactions and blocks are referenced by their respective ID.

2.2.1 Consensus Rules

Bitcoin's decentralized approach requires the community to establish consensus over the state of the blockchain. A wrong state could reduce trust in the currency and cause financial harm to all participants of the network: this incentivizes every participant to keep the blockchain in a valid state.

Miners are the ones who generate new blocks and hence, need to verify their correctness. The generation of a block includes a computational puzzle, called *proof-of-work*, to make the block valid. In the specific case of Bitcoin, miners have to *find* a random value (or nonce) and add it to the block header, so the BID has a certain format. The difficulty of the puzzle is adjustable and is set by the Bitcoin protocol such that new blocks are mined approximately every ten minutes. This is an important factor for its current scalability issues, as discussed in Section 1.1. Miners are rewarded for their work by charging fees

[2] Except the founding transaction

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

and creating new coins (hence the term “mining”), however, if they fail to follow the rules their blocks will be ignored by other miners, and their earnings are rendered useless.

It may happen that two independent miners add a new block at merely the same time, thus creating two versions of the chain (one with block A and one with block B): this scenario is called a *fork* of the blockchain. If both blocks are correct, new ones could be added on top of either one. It depends on the miners on which version they continue, however, in the end, the community accepts only the longest correct version of the blockchain.

Forks make the top of the blockchain rather unstable, as it is unsure which version miners will choose to continue on. Hence, to be on the safe side, one usually waits until a block is buried deep enough on the longest chain before acknowledging it. A block is said to have received n confirmations, if there are n blocks added on top of it. Typically, after being confirmed 6 times it is safe to assume a block is part of the blockchain. As block mining takes on average ten minutes, payment confirmations can take up to one hour (slow compared to standard online banking).

On a higher level, we can derive the following conditions to be met for a block to be considered valid and ready to be included in the blockchain:

(1) The BID is computed correctly (according to the *proof-of-work* puzzle)

(2) The TXID of each transaction is computed correctly

(3) There are no contradicting transactions in the block (e.g., spent the same transaction output)

(4) Each input of a transaction references only outputs that have not been spent before (no double spends)

(5) Each transaction does not spend more coins through their outputs as are available on its inputs[3]

(6) Each input’s script satisfies the referenced output’s script (i.e., evaluation is successful)

(7) Absolute and relative timing constraints are met by each transaction and their inputs (see Section 2.2.3)

Points (1) and (2) can be verified by recomputing the IDs. To exclude double spending as required by (3) and (4) one has to parse the whole blockchain and check if some output is referenced already. In practice, one would parse the blockchain once and create the UTXO-set, containing references to all unspent outputs and when new blocks are added the set is simply updated. Concerning (5), all referenced outputs are checked and the

[3]Expect so-called *coinbase* transactions added by miners to create new coins.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

sum of their transferred coins is compared to the amount of coins spent by the current transaction. Through (6) one can define conditions for how their coins can be spent, while (7) provides a way to direct when a transaction should be added to the chain. By combining both, it is possible to create time-dependent output *locks* [Pet14, Mar15a]. The semantics of Scripts and timing mechanisms of Bitcoin are presented in the following two sections.

2.2.2 Scripts

As mentioned before, Bitcoin utilizes simple programs written in `SCRIPT`, a custom stack-based scripting language, to identify who is eligible to spend a transaction output. A script consists of a list of opcodes that are interpreted as a series of operations. The language supports operations for cryptography such as hashes and signature verification, stack manipulation, arithmetic, boolean logic, and branching, but doesn't provide looping instructions, which makes the language non-Turing complete. A list of opcodes and their meaning can be found in [Bit24b].

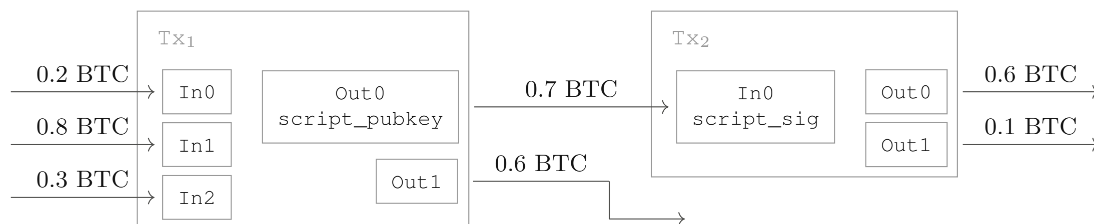


Figure 2.1: Illustration of two Bitcoin transactions

The originator of a transaction (payer) includes a script for each transaction output to specify how coins can be redeemed. The output script is, however, only one part of the whole script. The receiver of the payment has to specify with each transaction input a second script, including the information that satisfies the output script. Commonly, the output's script is referred to as `script_pubkey` and the input's script as `script_sig`. Figure 2.1 depicts two transactions Tx_1 and Tx_2, with the only input of Tx_2 referring the first output of Tx_1 (thus moving coins from Tx_1 to Tx_2 in the direction of the arrow). The figure explicitly highlights the scripts of the involved entries. Miners append the input script in front of the referenced output's script and evaluate the concatenation (i.e., `[script_sig++script_pubkey]`). Thereby, the input script is executed first and sets up the stack with the data required for the output script to run successfully.

In the Bitcoin network, the identities of individuals are hidden behind public keys, which are often referred to as Bitcoin addresses. To transfer coins to a specific receiver, one of its public keys (or address) has to be known. The output script performs a signature verification, whereas the signature is provided through the input scripts and has to match the public key. As signature generation requires a secret signature key, it is ensured that the coins can only be redeemed by the individual in possession of that key. Some applications require the authorization of multiple principles to conduct payments.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
script_pubkey := [OP_DUP ; OP_HASH160 ; OP_PUSH <hash160 public-key> ;
    OP_EQUALVERIFY ; OP_CHECKSIGVERIFY]

script_sig = [OP_PUSH <public-key> ; OP_PUSH <valid-sig>]
```

Listing 2.1: Pay-2-Pub-Key-Hash script

For such scenarios SCRIPT, supports multi-signature checks [Gav11]: the input script has to contain more than a single valid signature (bounded by their public key) to redeem an output. Those outputs that include multi-signature checks are said to send coins to a *multi-sig* address.

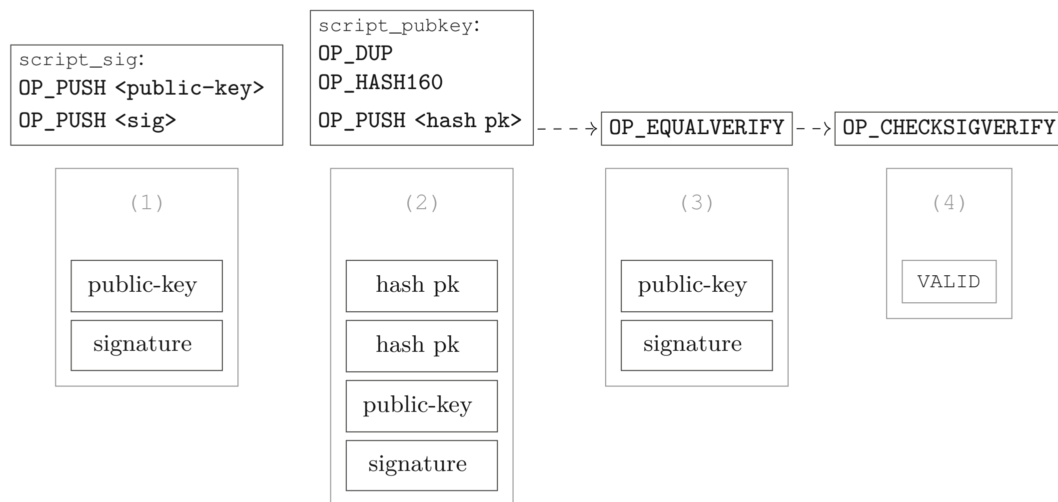


Figure 2.2: Stack evaluation of Pay-To-Pub-Key-Hash script on valid input script

As an example, Listing 2.1 shows the standard output script Pay-2-Pub-Key-Hash, together with the associated input script, which specifies the hash of a public key belonging to the receiver (OP_PUSH <hash160 public-key>). The receiver has to provide the original public key together with a valid signature in order to redeem the coins. Scripts evaluation hashes the public key from the input scripts and compares it with the hash in the output script before verifying the signature with said key. Other common scripts are the Pay-2-Pub-Key script and payments to multi-sig addresses. The evaluation of the scripts in Listing 2.1 is shown in Figure 2.2, divided into four phases: each phase shows the script part that was executed and the resulting stack below. The opcode OP_PUSH <data> pushes the defined bytestring on the stack, OP_DUP duplicates the top value of the stack, OP_HASH160 removes the top of the stack hashes it and puts the result back on the stack, OP_EQUALVERIFY removes the two topmost values and continues only if they are equal and finally OP_CHECKSIGVERIFY pops a public key and a signature (relating to the transaction [Joh16]) from the stack and verifies the signature given the key.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2.3 Timelocks

Timelocks can be used to postpone the validity of transactions to a later point in time. Bitcoin implements two timelock mechanisms to express **absolute** and **relative** timing constraints. They are controlled by special fields inside the transaction definition and are set during creation: each transaction has a field `nLocktime` and each input a separate field `nSequence` [Mar15b]. Miners have to check both fields and identify if they mine the transaction or reject it for now:

- Setting `nLocktime` to height `h` tells the miners to reject the transaction until the blockchain contains at least `h` blocks (**absolute timelock**).
- In contrast, `nSequence` lets one specify the *age* of an output referenced by a given input (**relative timelock**). When setting `nSequence` to height `d`, the corresponding input is only valid once its referenced output is buried by at least `d` blocks (i.e. the block containing that output has `d` confirmations).

It is also possible to use Unix-time instead of block height, but since it is rarely used, we will omit this feature for the rest of this document.

The scripting language defines two opcodes for comparison of the timelock fields within scripts: `OP_CHECKLOCKTIMEVERIFY` (CLTV) and `OP_CHECKSEQUENCEVERIFY` (CSV). Scripts including either opcode impose timing conditions on redeeming transactions, such as timeout periods (see Section 2.3). Both opcodes have been introduced through soft-forks[4] [Pet14, Mar15a] and replace `NO_OP` operations (which leaves the stack as it is) such that Bitcoin clients running on an older version simply ignore the new commands.

During script evaluation, the interpreter has access to `nLocktime` and `nSequence`. Both fields need to be set accordingly to satisfy the output script, otherwise, the evaluation fails and the transaction is rejected. If they are set correctly and the evaluation is successful, it is guaranteed by consensus that the timing constraints are met. Indeed, if a (malicious) principal tries to consciously alter a timelock field (e.g., to skip timeout periods) the script evaluation will fail when the timelocks are compared.

[4] A soft-fork is a way of adding features to the Bitcoin protocol while ensuring older versions still function correctly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Transaction Headline			
Input	**Output**		**Signed By**
0. Tx_1	**0.** v **coins to:**	`Alice`	Alice
	1. x **coins to:**	`Alice` after t blocks	
	2. y **coins to:**	`Alice` at height h	
	3. z **coins to:**	`Alice` if M	

Figure 2.3: Example transaction to show notation used in this section

2.3 The *Lightning Network*

The section introduces the off-chain PC implementation of the LN [PD16, LN 16], and multi-hop payments with HTLCs in more detail, highlighting the aspects relevant to this thesis. This section and the illustrations in it (regarding transaction formats) are inspired by the lecture of [ND18] and the original white paper [PD16]. It is structured as follows: First, we briefly introduce the idea of simple bidirectional *Payment Channels* and show how transaction revocation is done through punishing dishonest principals. We then move on to transaction formats, that represent immediate channel states. Afterwards, we proceed to dive deeper into HTLCs and, finally, combine simple channels with HTLC outputs to obtain more advanced networked channels with payment forwarding.

Within this section, simplified Bitcoin transactions are depicted to help understand the concepts presented (see Section 2.2 for a description of Bitcoin transactions). For simplification, we only consider four types of output conditions (defined by output scripts) shown in the example transaction in Figure 2.3. The depicted transaction has only one input Tx_1 and four outputs that spend v, x, y, and z coins from the input transaction. Furthermore, the transaction has been signed by `Alice`, meaning she agrees with its contents. **Output 0** can be redeemed with `Alice`'s signature right away. If more than one signature is required (i.e., multi-sig address) we denote this with `A + B`, meaning a valid signature of `A` and `B` is required. **Output 1** can be redeemed with `Alice`'s signature after this transaction received t confirmations (relative timelock). **Output 2** can be redeemed with `Alice`'s signature when the blockchain has reached an absolute height of h blocks (absolute timelock). **Output 3** can be redeemed with `Alice`'s signature together with the value M. The logical combination of conditions is possible.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.3.1 Simple Payment Channels

Payment channels enable the exchange of several micro-transactions that happen off-chain, avoiding to overload the blockchain. The visible footprint of a PC consists, ideally, of only two blockchain entries, a *funding* and a *closing* transaction. The former establishes a shared budget between two principles, `Alice` and `Bob`, and the latter distributes that budget between them, according to their last state of consensus (see Figure 1.1).

To open a channel, at least one of them must transfer an initial amount of coins to a multi-signature address, i.e., signatures from both participants are required to redeem the shared coins. Now they can create *commitment* transactions (which are valid on-chain transactions), referencing the funding transaction to change the current distribution of coins. In other words, a commitment transaction defines a new channel state.
Since the funding transaction is controlled by both `Alice` and `Bob`, they both have to agree on new states, by signing the commitments. Each party creates its own version of a commitment transaction, signs it, and sends it to its peer to store it (e.g., on their own storage device). Once the commitments are exchanged, payments are valid immediately as either party can add the missing signature instantly and broadcast it to the network. They can create and exchange as many commitment transactions as they want. When they are done, only the last one is published on the blockchain. This closes the channel.

Transaction refunds. The creation of the funding transaction requires special care. Once published the referenced coins cannot be retrieved unless both parties cooperate. However, if one party acts maliciously, they may keep the coins locked by refusing to sign additional commitments.
Therefore a *refund* transaction is established before signing the funding transaction. This refund transaction will reimburse `Alice` and `Bob` with their shares of the initial channel budget. It is important to note that the refund transaction references the funding transaction, which is, at this point, unsigned and unpublished. Prior to SegWit [Eri15], the signature affected the calculation of the TXID, hence it was not possible to generate the TXID of unsigned transactions.
If the funding transaction is signed first, there is again no guarantee that both peers cooperatively proceed to create the refund afterwards.

Transaction revocation. Every commitment transaction of an open channel is valid and can be broadcast anytime. Hence, a malicious node may choose to publish an outdated, but financially more favorable, transaction, which will eventually be accepted in the ledger.

It is important to note that published transactions can not be undone once they have been confirmed and entered into the blockchain, i.e., it is not possible to remove them. The revocation mechanism, therefore, involves an additional transaction, undoing the effects (movement of coins) of wrongfully entered ones. To make the revocation possible, whenever new commitment transactions are created, old ones are invalidated, by having `Alice` and `Bob` exchange private *Revocation Keys* [ND18, LN 24]. In the following, we

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Commitment Tx (created by Alice)			
Input	**Output**		**Signed By**
0. Funding Tx	**0.** x **coins to:**	Alice	Alice
	1. v **coins to:**	Bob after t blocks ∨ Alice + rk_bob	

Figure 2.4: PC Commitment Transaction created by Alice

denote the *Revocation Keys* created by Alice and Bob, with rk_alice and rk_bob respectively. The keys are used to unlock a special output of the commitment transaction, that allows one peer to acquire all coins, locked in the funding transactions, in case their dishonest counterpart tries to close the channel with an invalidated commitment. They can spend the output by proving they are in possession of the corresponding private key, i.e., presenting a valid signature (as part of script_sig). A transaction transferring coins from that "punishment output" is often called a *Breach Remedy Transaction* (BRT) and is meant to punish dishonest behavior. This punishment acts as deterrence to render the attack unprofitable.

We want to emphasize an important point: the keys must initially be secret, but they can be made public when creating further commitments. Since Alice and Bob can not be sure that their counterparty deleted old transactions, the exchange of rk_alice and rk_bob acts as insurance for both to not get defrauded in the future.

Commitment Transaction Formats

The format of the commitment transactions created by Alice is shown in Figure 2.4. The transaction is partially signed by Alice and defines a new state, granting x coins to Alice and v coins to Bob, with $x + v$ being the total amount of their shared budget. There is only one input, the funding transaction, and two outputs. **Output 0** guarantees x coins to Alice right away (bounded to her signature). **Output 1** has a condition, which says that v coins go to

(1) Bob if more than t blocks are added after the block holding this transaction (**relative timelock**) or

(2) immediately to Alice if she knows rk_bob (presenting her signature and one using rk_bob).

Alice now hands over this partially signed transaction to Bob, who could add his signature and publish it, thereby closing the channel, or hold onto it indefinitely.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Commitment Tx (created by Bob*)*			
Input	**Output**		**Signed By**
0. Funding Tx	**0.** x **coins to:**	Alice after t blocks ∨ Bob + rk_alice	Bob
	1. v **coins to:**	Bob	

Figure 2.5: PC Commitment Transaction created by Bob

Since Alice already signed the transaction, Bob could wait and publish it at a later point in time, even if the state in this commitment is outdated. In such a case the condition of **output 1** acts as a safeguard to Alice. If she is in possession of rk_bob, she can claim all the coins in the funding transaction (thereby punishing Bob). Ideally Alice knows rk_bob, as they should have exchanged their private keys for old commitments a priori. Still, Alice has to monitor the blockchain, in order to act quickly enough before Bob's timelock expires.
Output 0 specifies the funds reserved for the counterpart, in this case Alice. There is no condition to this output, as by broadcasting, Bob agrees to grant x coins to Alice.

Bob's commitment transaction is shown in Figure 2.5. It has a similar structure to Alice's, but now punishing Alice. **output 0** requires her to wait for t blocks to redeem her x coins. If the state is old, and Bob knows rk_alice, he can claim this output before Alice's timelock expires. Bob can claim his v coins right away, via **output 1**.

2.3.2 Multi-Hop Payment Channels

The creation of PCs is only practical if multiple micro-transactions will take place. Otherwise, the overhead of opening and closing a channel is too large. This, however, implies that one-time transactions still have to be carried out over the blockchain, despite limited transaction throughput. Forming a channel between every peer on the network is also not desirable, making PCs alone, not a scalable solution.

Having to open new channels for one-time payments is not very economical: at least two transactions have to be published, and besides the confirmation waiting time, fees to mining nodes incur. The economic aspect is even worse for small payments.
Instead of opening new channels, it would be far more efficient to utilize existing ones. With multi-hop (or networked) channels, transactions can be forwarded, through multiple intermediate nodes, required one can find a route of actively open channels (i.e., each node on the route is responsive). The forwarding is done by adding a new output type to the already existing commitment transactions, namely a *Hashed Timelock Contract*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Hashed Timelock Contract

An HTLC is a regular output script, with a special definition [LN 24, Sea17, PD16]. It is not tight to use for PCs only but can be included in any transaction output if desired. One can see it as a contract between two parties `Alice` and `Bob`, according to the following conditions:

(1) `Bob` gets coins from `Alice` if he knows the preimage `R` of some hash `H(R)` specified by `Alice`, or

(2) `Alice` is able to refund the coins at a specific time point (**absolute timelock**).

To fulfill the contract `Alice` requires a *proof-of-knowledge* by `Bob`, otherwise if `Bob` is not able to present the preimage until the set deadline, `Alice` gets her coins back.

We can write down an HTLC in the form of a constructor

`htlc(A,B,y,h,H(R))`,

where principal `A` pays y coins to principle `B`, if `B` learns `R` before the blockchain reaches height `h`.
Note that HTLCs can be used to create conditional payments. In Figure 2.4 **output 1** guarantees v coins to `Bob`, as long as this commitment stays valid. From `Alice` point of view, these v coins are gone for now, and can not be taken back. An HTLC output, on the other hand, places a condition on `Bob`'s side, to deliver a special value, before he gets paid. In this case `Alice` can revoke the payment, if `Bob` does not succeed.

Multi-Hop Payments

As already mentioned before, multi-hop payments utilize existing PCs to forward payments through a network of channels. However, if `Alice` sends coins to `Bob` and asks him to forward the coins to `Carol`, via an existing channel, there is a likelihood he simply keeps the coins to himself. This is because simple channels lack the ability to put any obligations on channel participants on how to spend their funded budget. To make sure `Alice` does not lose coins to `Bob`, in the case he does not comply, `Alice` could set up a contract, more specifically an HTLC.

The contract promises `Bob` he will be reimbursed by `Alice` if he can prove he sent the agreed amount of coins to `Carol`, otherwise `Alice` will not pay him. As an incentive for `Bob` to cooperate `Alice` may choose to give him a fee for his service. The *proof* is a secret value, only known to `Carol`, but verifiable by `Alice` (i.e., the preimage of a hash). When `Bob` pays `Carol`, she reveals the secret to him.

This payment process is illustrated in Figure 2.6, assuming PCs between `Alice` and `Bob`, as well as `Bob` and `Carol` exist. Solid lines refer to the creation of HTLCs with the given parameters, whereas the arrow starts at the peer, who initiated the contract, while dashed lines represent communication outside of any channel. The figure shows the 5 steps to establish and settle the forwarded payment (`Alice` giving y coins to `Carol`):

TU Bibliothek Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
WIEN Your knowledge hub The approved original version of this thesis is available in print at TU Wien Bibliothek.

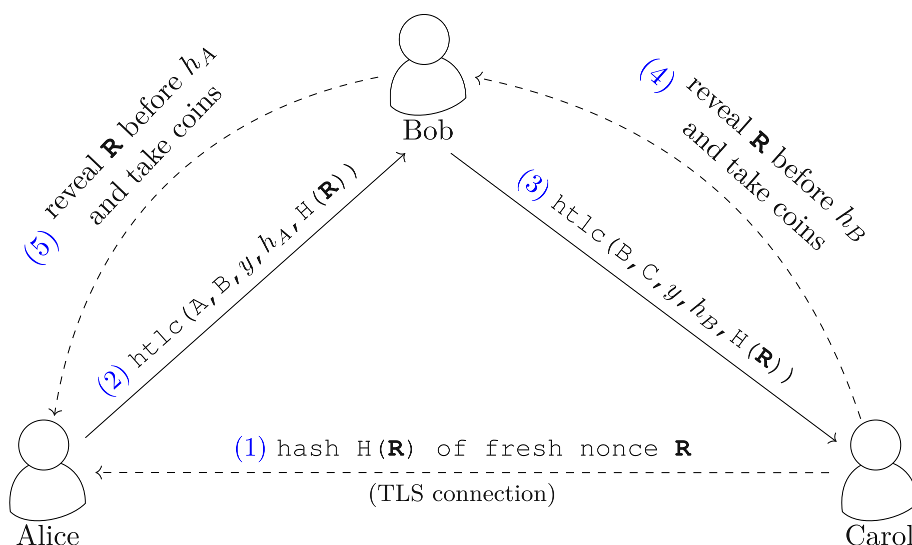


Figure 2.6: Successful Multi-Hop payment with HTLCs

1. `Carol` generates a fresh secret `R` and shares the hash `H(R)` with `Alice`
2. `Alice` creates an HTLC between her and `Bob` over `H(R)` with deadline h_A
3. `Bob` creates an HTLC between him and `Carol` over `H(R)` with deadline h_B
4. `Carol` settles the contract by redeeming y coins from `Bob` and **disclosing R** to him
5. `Bob` settles the contract by redeeming y coins from `Alice` and **disclosing R** to her as proof he complied to the contract

It is important that `Bob` sets $h_B < h_A$, otherwise `Carol` and `Alice` can close the channels at the *same* time (when h_A runs out), stealing coins from `Bob`. Also, note that multi-hop payments can be conducted with an arbitrary number of intermediate hops.

To support multi-hop payments via simple *Payment Channels*, the new HTLC output is appended to the original commitment transactions. Note, that the HTLC is not present all the time, it is only added when needed. Figure 2.7 shows `Alice`'s updated commitment transaction with payment forwarding. **Output 2** is the so-called *HTLC-output*, setting up the contract between `Alice` and `Bob`. **Output 0** and **1** define the current state of funds distributed between `Alice` and `Bob`, whereas `Alice` gets y coins less than in the previous state. **Output 2** states that `Bob` can spend those y coins if he knows the hash preimage `R`, or `Alice` can claim them back, once the timelock expires (i.e., the blockchain reached absolute block height h).
Figure 2.8 shows `Bob`'s new commitment transaction containing the HTLC-output. The output is similar to `Alice`'s, if `Bob` wants to take the y coins he can only do so if he knows `R`, and `Alice` can revoke the payment after height h.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Commitment Tx with HTLC (created by `Alice`*)*		
Input	**Output**	**Signed By**
0. Funding Tx	**0.** $x - y$ **coins to:** `Alice`	Alice
	1. v **coins to:** `Bob` after t blocks ∨ `Alice` + `rk_bob`	
	2. y **coins to:** `Bob` if preimage of `H(R)` ∨ `Alice` at height h	

Figure 2.7: Commitment Transaction with HTLC output created by `Alice`

Commitment Tx with HTLC (created by `Bob`*)*		
Input	**Output**	**Signed By**
0. Funding Tx	**0.** $x - y$ **coins to:** `Alice` after t blocks ∨ `Bob` + `rk_alice`	Bob
	1. v **coins to:** `Bob`	
	2. y **coins to:** `Bob` if preimage of `H(R)` ∨ `Alice` at height h	

Figure 2.8: Commitment Transaction with HTLC output created by `Bob`

The HTLC output (e.g., Figure 2.7 output 2) uses absolute timelocks since the timeout should expire regardless of the transaction being broadcasted or not. In contrast, the *revocable* output (e.g., Figure 2.7 output 1) has a relative timelock to give honest peers enough time to react after the transaction has been maliciously published (independent of the time the transaction enters the blockchain).
HTLC payments are done within two rounds of communication: (1) new commitment transactions containing the HTLC-output are created, and then (2) debts are settled by either presenting `R` or issuing a refund after the timelock.

Note that revocation of HTLC outputs requires several pre-signed transactions, exchanged by `Alice` and `Bob`, which have not been discussed here (e.g., *HTLC-Timeout* and *HTLC-Success* transactions). The idea is similar to "standard" revocable outputs, which means having separate signature keys for every possible execution branch. For more details, we refer the reader to [PD16] and [LN 24].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Closing a Multi-Hop Channel

In a *best-case* scenario, honest and responsive peers can handle many steps without having to rely on commitment transactions. For example, when `Carol` is revealing `R` to `Bob`, they may agree to clear the HTLC-output from their channel (since obviously `Carol` can take her coins if the current commitment transaction of their channel is published) and create a commitment with a new balance, guaranteeing her the received coins. If `Bob` is not responding, `Carol` has to publish the transaction containing the HTLC, so, she can take her coins before `Bob` chooses to refund them (after the timeout). This, however, closes the channel.

In total a multi-hop channel can be closed in three different ways[5]:

- *Mutual Close:* If both parties act honestly and cooperate they can choose, to erase all unnecessary conditions from their last commitment transaction and create a new clean transaction, that simply sends coins of the funding budget to both parties, according to their last state.
  At least two transactions are added to the blockchain: *Funding Transactions* and a freshly created *Closing Transaction*.

- *Revoked Transaction Close:* An invalid commitment transaction gets published and is revoked afterwards, punishing the broadcasting node.
  At least three transactions are added to the blockchain: *Funding Transactions*, *Invalid Commitment Transaction* and *Breach Remedy Transaction*.

- *Unilateral Close:* One party could simply publish the last valid commitment transaction, containing all three conditional outputs. They might choose so if their peer does not respond or does not adhere to the protocol. Closing the channel this way limits the short-time liquidity of the publishing node (usually the one acting honestly), as a certain amount of time has to pass before an output is spendable.
  At least two transactions are added to the blockchain: *Funding Transactions* and the last valid *Commitment Transaction*.

Attacks on Multi-Hop Payments

The preimage `R` can be disclosed in two ways:

1. Both parties are cooperative and clear the HTLC-output by agreeing on a new state. In this case, `R` can be sent directly to the counterpart via a secure channel.

2. `R` is revealed via the blockchain embedded in the redeeming script (`script_sig`), by spending the HTLC-output.

[5]https://github.com/lightning/bolts/blob/master/05-onchain.md, accessed 27-April-2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

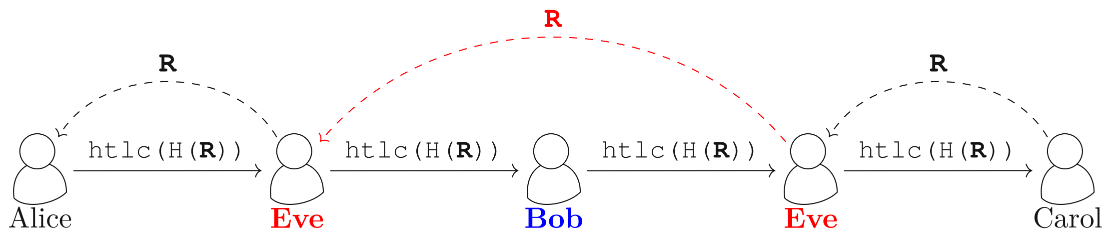


Figure 2.9: Simplified illustration of information leakage of Multi-Hop payments with HTLCs

In both cases, a malicious peer might get access to R, either directly by participating as a hop during payment forwarding or indirectly whilst monitoring the blockchain for spending of HTLC-output transactions (see the discussion on LN privacy in Section 1.1). Hence, one might want to treat R as a public value, once it has been revealed.
LN payment channels reuse the same preimage on every HTLC on the payment route, thereby compromising *Atomicity* requirements of the locking mechanism. Execution of the *Wormhole* attack [MMSS+19] on the LN takes advantage of the intended leakage of R, by bypassing intermediate hops along a payment route. The attack incorporates forwarding of the learned preimage by principle A to a cooperating hop B, closer to the sender of the payment. Thereby, A and B can collect payment fees, intended for nodes that lie between them on the route.

Figure 2.9 shows a simplified instance of the attack presented in [MMSS+19] (timelocks, coins, etc. were omitted for clarity). It instantiates a multi-hop payment from Alice to Carol that goes through Bob and twice through the attacker (Eve). To settle the payment, the recipient (Carol) sends R back the path to Eve. Then, Eve sends the preimage directly to Alice instead of Bob. From Bob's point of view, the payment has been unsuccessful. So he has to wait for the end of the HTLC's timeout for the channels on both sides to revert to their original state, losing the fees he was promised along the way.

Technically speaking, no financial harm is caused in this process. From the perspective of the sender, receiver, and non-bypassed intermediate nodes, the payment has been successful. Bypassed nodes perceive the payment to have failed and simply refund their stake. *Balance Security* [MMSK+17] still holds, as no participant of the payment path, other than the sender, effectively loses coins (not considering the loss in service fees). However, ideally, there should be an overall consensus on the payment outcome along the path, referred to as *Atomicity* [MMSK+17].

In addition to stealing service fees, Eve can deduce information about participating peers. Since every HTLC along the path is locked with the same hash H(R), Eve knows that Alice, Bob, and Carol are participating in the same payment. Using different hashes for each lock (involving higher communication effort), or replacing the HTLCs in favor of AMHLs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Type System of DY*

This chapter aims to provide the reader with a deeper dive into how DY* works and how it is used to provide meaningful security guarantees. It highlights the most important parts of the verification procedure, without going too deep into implementation details (refer to [BBD+21b, BBD+21a] for an extensive introduction), but accurately enough to understand where and why changes were made throughout this thesis.
DY* ("*Doley-Yao-Star*") is a framework to symbolically verify cryptographic protocols in the presence of a *Dolev-Yao Attacker* [DY83]. Such an attacker is able to intercept, modify and forward any message sent over the network, and apply any function to terms it has knowledge of. Hence, the attacker can interact with principals based on publicly available data. Additionally, in DY* the attacker can compromise protocol sessions of principals and thereby recover private information.

This chapter is structured in the following way: First, we introduce the concept of *global trace* and see how the underlying labeling system is constructed. Afterwards, we define what a valid trace is, providing a quick overview of F*'s effect system and explaining how it is used throughout DY* for labeled trace operations. Then we provide an introduction to DY*'s trace-based reasoning in the form of *witnessed* predicates. Finally, we present the implementation and first results of a labeling extension for dynamic leakage of secret data: the *releasable* label.

Throughout the sections, code snippets and type definitions, written in F* syntax, are shown or placed side-by-side with textual information to link theory and implementation. Note that those examples are taken from the implementation code of DY*[1] but may have been adjusted and simplified to explain the concepts.
F* features type refinements over any type `t`, written as `type t' = x:t{phi}`, where `phi` is some proposition, restricting possible values of `x`. The refinement creates a subtyping relation `t' <: t`, requiring `phi` to hold, whenever an element of type `t'` is expected,

[1]https://github.com/REPROSEC/dolev-yao-star, accessed 30-April-2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
type entry_t =
  | RandGen: b:bytes → l:label → u:usage → entry_t
  | SetState: principal → v:version_vec → new_state:state_vec → entry_t
  | Corrupt: corrupted_principal:principal → session:ℕ → version:ℕ → entry_t
  | Event: sender:principal → event → entry_t
  | Message: sender:principal → receiver:principal → message:bytes → entry_t
```

Listing 3.1: The type `entry_t` of global trace entries

otherwise verification will not succeed.
Type refinements are a form of *intrinsic* proofs in F* which abidance is automatically checked during verification. *Extrinsic* proofs, on the other hand, are explicitly stated as a `Lemma`, which can be recalled to "remind" F* of a fact that has already been proven. For a more detailed description of F* syntax and typing capabilities, we refer the reader to the online tutorial [Nik24].

3.1 Global Trace

The global trace logs all the operations that occur during a single protocol execution. In principle, it represents a timely-ordered list of each step performed by all principals involved. It introduces a notion of time by giving each entry in the trace a unique timestamp, representing its index in the trace (`type timestamp = ℕ`). By using F*'s proof machinery together with z3, queries about trace (or safety) properties can be conducted. In Section 3.5, we explain details about DY*'s trace-based reasoning.

The trace is defined as a list of trace entries, which includes, in the original framework, five different protocol operations. The corresponding type `entry_t` is shown in Listing 3.1. The definition contains entries referring to the generation of fresh random secrets (Line 2: `RandGen`), updates of principals state (Line 3: `SetState`), the triggering of protocol-specific events (Line 5: `Event`), and the transmission of messages (Line 6: `Message`). Additionally, the trace can contain entries to denote the corruption of one principal's sessions (Line 4: `Corrupt`) and the leakage of the associated data.

Figure 3.1 illustrates how the state evolves from a starting trace `t0` to `t1` after a protocol-specific event `A` is triggered. The figure depicts, in a simplified way, the latest entries of both traces. Trace indexing starts at timestamp 0, hence the length of `t0` is t. We say that the current time of a trace equals the trace length, hence `t0` is currently at time t. The new protocol step always happens at the next available timestamp; therefore, when event `A` occurs, it gets triggered at time t, corresponding to the insertion index. The execution of the event trigger causes the trace to evolve from `t0` into `t1`, and since all the entries up to $t-1$ are equal, we have that

```
t1 = (Event A)::t0.
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

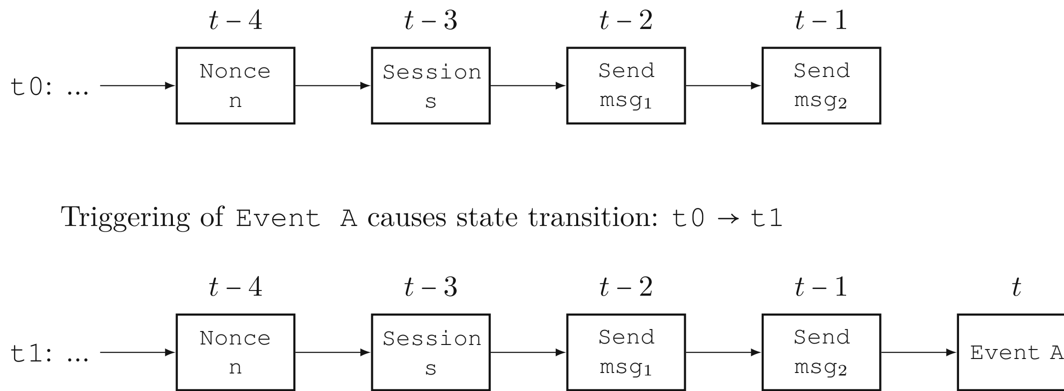


Figure 3.1: Evolution of the global trace after triggering an event

The attacker can call all functions available but is restricted to using only data derived from its current knowledge by applying primitive functions to public data (i.e., literals or data sent over the network). The attacker can split tuples of values, reduce them with known functions, and obtain new keys from a given context. Public data is made available to the attacker when sent over the network. (e.g., each trace entry of the form `Message:sender:principal → receiver:principal → message:bytes` contributes to the attacker's knowledge.)

The global trace definition allows verification against all possible protocol executions still, the trace is never explicitly built. DY* interprets the trace as a logical construct, which adheres to certain invariants. Standard invariants, like secrecy of private values, are enforced by the framework, while protocol-specific rules are defined by the user and are written down as trace invariants and usage predicates (more information can be found in Section 3.3).
Functions that alter the trace are verified to adhere to the specified rules, from which properties can be concluded. This technique demands some effort when designing and proving postconditions of lemmas, but eases type-checking in the long run. Also, properties have to be proven only once and can be used later on, without additional compiler effort.

Most of the benefits of using DY* come from its API, which abstracts away the trace from the protocol designer while helping F* to prove that the trace invariants are indeed upheld. The API includes functions for

- creating new and recalling old session states,
- sending and receiving messages on the network,
- generating random data (nonce),
- retrieving public and private key material,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

- triggering events (to ensure authentication and restrict protocol flow), and
- all sorts of cryptographic operations, such as encryption signatures and hashes.

Writing protocol code can be done entirely using API calls, thus handling the trace directly is not necessary.

3.2 Labeling System and *Can Flow* Relation

Similar to other security type systems (e.g., [FM11, BHM14]), DY* defines labels to annotate objects and restrict the data flow, using the type `label`. The subtyping relation of labels is defined through a lattice given the `can_flow` predicate[2]:

```
val can_flow:(ts:timestamp)→ (l1:label)→ (l2:label)→ Type0
```

The relation takes a timestamp and two labels and returns a proposition stating whether `l1` can be subtyped as `l2` at timestamp `ts`. In DY*, labeling is *static*, meaning that once the label of an object is set, it cannot be changed at a later timepoint. An exception to this rule is the state-compromising feature of the attacker, allowing secrets to be type-checked as public values if the state is compromised, thus changing their behavior.

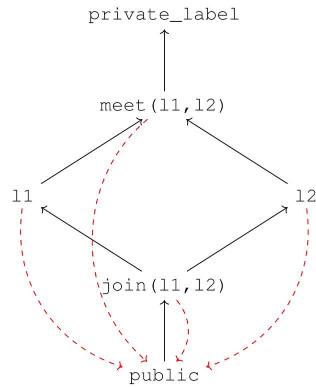
Secret values are typed with the label `val readers (list id):label`, denoting that only principals whose identifier is in the list of possible readers are allowed to access the value. The top element of the labeling lattice is the type `let private_label:label = readers [ ]`, referring to data that can not be read by any principal (i.e., an empty list of readers). Every label can flow to the top element, as restricting the number of possible readers is always safe. The bottom element, instead, is `val public:label`, indicating a value that can be read by everybody. This label is given to types that can be safely sent over the network or that belong to a compromised principal's state. The lattice is completed by the labels `val join:(l1:label)→ (l2:label)→ label` and `val meet:(l1:label)→ (l2:label)→ label`, which represent the union and intersection of labels respectively.

Nonces (i.e., random numbers) can be given arbitrary labels, while constants (represented by string, integers, or bitstrings) are always `public`. The `meet` and `join` operations let us assign labels to composite terms such as tuples, enums and cryptography-related objects.

Most of DY*'s modules don't have access to `can_flow`'s implementation for verification (for modularity reasons, F* restricts access to one module's implementation), but they instead rely on a set of lemmas that captures the relation's behavior. The properties of `can_flow` can be split into two categories:

[2]Note that `Type0` is used to encode propositions that are not decidable (e.g., function equivalence) and have to be proven using the SMT solver. In contrast, `bool` defines solely decidable equality.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub



Legend:

- `l` ⟶ `l'` denotes `can_flow p i l l'` (assuming a reflexive and transitive graph).
- `l` - - → `l'` indicates the effective new label of `l` after corruption.

Figure 3.2: Lattice order of *can-flow-relation* with corruption

- **Temporal Relations**:
  Defines how labels behave, depending on the current timestamp. For example: `can_flow_later t1 t2 l1 l2`, to ensure once a label `l1` can flow to `l2` at `t1` it can also flow at a later timestamp `t2`.

- **Label Flow**:
  This includes the general flow of labels according to their definition, as well as proofs of *reflexivity* and *transitivity*.

Given those properties, `can_flow` forms a preorder over the set of labels. DY*'s dynamic corruption system (as explained in Section 3.1) can eliminate the static specification and can cause any label at any timestamp to flow to any other label. Once the attacker learns a secret, it is not safe anymore to type it as private: instead, it has to be treated as public and the rest of the protocol implementation has to take the possibility into account. Thus `can_flow` is not a total order, or even an order at all, as shown in Example 3.2.1.

Example 3.2.1 (`can_flow` is not a total order) *Note that it is not total: for example, the two labels `readers [P a]` and `readers [P b]` of two different principals `a` and `b` can not be ordered (i.e., one is not a subtype of the other).*

It is not an order either: for instance, once `ids` contains a corrupted agent, `readers ids` flows to `public` and vice versa; hence the lack of anti-symmetry.

In Figure 3.2 the simplest lattice ordering for two non-comparable labels `l1` and `l2` is shown, which illustrate the position of `meet` and `join`. The black arrows define the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
val pke_enc: #i:timestamp → #nl:label → public_key:msg i public →
    nonce:msg i nl → message:msg i (get_sk_label public_key){
      can_flow i (get_label message) nl ∧
      can_pke_encrypt i public_key message} →
    msg i public
```

Listing 3.2: Simplified signature of the public-key encryption function `pke_enc`

direction in which labels can flow, while the red arrows indicate the corruption of labels. Each label can at any point in time flow to `public`, once any of its principals has been corrupted (thus reverting the original label flow). The `meet` of two labels is semantically comparable to the intersection of both reader lists, hence it restricts access. In contrast, `join` grants access to any of its two label arguments.
The figure, of course, does not cover all cases possible with two labels, as there are infinite options. With each set of two labels, we can create a new one using `meet` or `join`, which can be used to create yet further labels, and so on. Even if the label `meet (meet(l1,l2),l1)` is semantically not different from `meet(l1,l2)`, syntactically they are not the same. DY⋆ does not actually compute `meet` and `join` into a set of eligible readers, but instead defines their semantics through `can_flow`.

To demonstrate how `can_flow` is used to define typing relations (as known from formal type system definitions) and to enforce security guarantees like secrecy, we take a look at the (simplified) declaration of public key encryption `pke_enc`, shown in Listing 3.2. The type `msg i l` is given to values whose label can flow to label `l` at time `i` (i.e. can be typed with label `l` at `i`). The encryption function takes as argument a public key of type `msg i public` (hence it can be typed `public`), a nonce of type `msg i nl`, and the message to be encrypted. Public values are considered *publishable*, and hence it is ensured that public keys can be sent over the network and shared with other principals (including the attacker).
The type of the message is a bit more interesting: it is also of type `msg p i l`, but (1) `l` is expected to equal the label of the corresponding secret key of the given public key (`get_sk_label public_key`), thereby making sure the principal in possession of the secret key is eligible to read the encrypted message and (2) the type is given a refinement, written inside the curly braces (Line 3 and Line 4). The defined refinement enforces restrictions over the message to be encrypted: the label of `message` (and hence also the label of the secret key) has to be a supertype of the nonce-label `nl` (Line 3) and the public key is eligible to encrypt the message at `i` (Line 4), verified using the usage-predicate `can_pke_encrypt` (see Section 3.3). Overall, this refinement guarantees that public key encryption is performed only with valid keys and messages can be safely typed on the receiving side.

If the message consists of several values gathered into a tuple its label is given by the `meet` of all individual item labels (which requires nesting `meet` definitions if the tuple

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
val send: (#i:timestamp) → (sender:principal) → (receiver:principal) →
    (message:msg i public) → ...
```

Listing 3.3: Signature of send function with label information on message

```
1 let valid_trace (tr:trace) =
2     (∀ (i:timestamp) (t:bytes) (s:principal) (r:principal).
3         i < trace_len tr  ⟹
4         (was_message_sent_at i s r t  ⟹  (is_publishable i t)))
5     ∧ (∀ (i:timestamp) (p:principal) (v:version_vec) (s:state_vec).
6         i < trace_len tr  ⟹
7         (state_was_set_at i p v s  ⟹  (state_inv pr i p v s)))
8     ∧ (∀ (i:timestamp) (s:principal) (e:event). i < trace_len tr  ⟹
9         (did_event_occur_at i s e  ⟹  (event_pred_at pr i s e)))
```

Listing 3.4: Definition of a valid trace

includes more than two values). As `meet` creates the semantic intersection of all labels, secrecy is given by the label lattice by restricting access to the tuple only to the readers in common.

To send an object with label `l` at timestamp `i` over the network, it has to be "publishable", meaning that `can_flow i l public` holds. In this context, publishable means that the message is allowed to be known by anyone (especially by the attacker, who can intercept any message sent). The signature of the `send` function, shown in Listing 3.3, only takes messages as input that can be typed `public`, i.e. have type `msg p i public`. The encryption function returns exactly the type needed, allowing to safely send secret data over the network, provided it is encrypted.

3.3 Valid Trace

DY⋆ enforces security properties via a typed protocol implementation style and verification of a set of trace invariants, including general statements (valid for any secure protocol) and protocol-specific invariants (only applying to a concrete implementation). A trace adhering to this specification is considered *valid*, and valid traces are identified by the `valid_trace` property of Listing 3.4. In the base implementation of DY⋆, we can identify invariants for three parts of the protocol: (1) Secrecy, (2) Sessions, and (3) Events Verification is performed by requiring a valid trace throughout each step of the protocol.

The first one, *Secrecy*, is a general invariant, that applies to all secure protocols. It is defined in Line 2 of the `valid_trace` predicate (Listing 3.4) and ensures that messages that have been sent at time `i` were publishable at that time. The clauses in Line 5 and Line 8

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
type usage_preds = {
  can_sign: timestamp → timestamp → key:bytes → msg:bytes → Type0;
  can_pke_encrypt:timestamp → timestamp → pub_key:bytes → msg:bytes → Type0;
  ...
}
```

Listing 3.5: Type of usage-predicates

take protocol-specific session state and event invariants into account and are specified by the user, called `state_inv` and `event_pred_at`. This enables reasoning over the protocol flow of execution. At the beginning of each protocol implementation, a set of reachable session states is declared, as well as a set of events, which are relevant to meet security goals (e.g., authentication). Then the trace invariants `session_st_inv` and `can_trigger_event` are defined over the set of available states and events: the former controls when and by whom a certain state can be reached, and the latter when events can be triggered (examples: *another event occurred before* - *a nonce, that is included in the event, has been generated before* - *a specific session state has been reached*).

In addition to trace invariants, usage predicates, defined in Listing 3.5, enable fine-grained control over the intended use of cryptographic material, like signatures (`can_sign`) and public key encryption (`can_pke_encrpyt`). A valid trace ensures that the respective usage predicate holds upon message encryption or signing (see example of public key encryption in Section 3.2). By making honest principals adhere to those predicates, one can recall them when doing the composite action. For example, if honest principal `A` signs a message and sends it to principal `B`, `B` can be sure that `can_sign` was satisfied by `A`. This allows one to infer judgments on the receiving side.

3.4 The CRPYTO and LCRYPTO Effects

Effects in F*. Effects define computations that differ from pure total functions, like mutable state or divergence. F* supports a rich system of computational effects, defined via a monadic representation with *weakest-predcondition-transformers* (WPT). The basis of effect theory and a formal definition of WPTs can be found in [SCL13, SHK+16, AHM+19]. A WPT is a function of the form `wpt: post → pre` that takes as input a postcondition and returns the weakest precondition needed to satisfy the postcondition. As effects are represented by monads, one needs to define combinators for computations (e.g., bind, if-then-else, return), as well as their WPT, that define the workings of the effect. F* uses those combinators to generate the necessary verification conditions (VC) to be discharged by z3.
The core of F* is a pure functional programming language, where each function is guaranteed to terminate. These functions are part of the `PURE` effect. F* predefines multiple effects on top of it to introduce side effects (the `Tot` effect, used by default) or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

non-terminating functions (the `Div` effect) and much more. Those effects are layered out in a lattice restricting when to call one from the other (e.g., one can call a `PURE` function from a `Tot` one, but not the other way around).

A function is defined by its effect type together with a return type (the result of computation) and a WPT. As writing down specifications using WPT can be difficult and hard to read, it is also possible to create sub-effects that take a pre- and a postcondition instead (similar to Hoare annotations). During type-checking, F* verifies that the implementation of the function matches its effect and proves its annotations. A strength of type systems is that this verification needs only to be done once. Afterwards, the concrete implementation is hidden behind its effect properties and never looked at again. Many side effects (e.g., error handling, divergence, mutable-state) are transparent to the end user due to the monadic definition.

DY* defines two effect types called `CRYPTO` and `LCRYPTO`, whereas the latter is a refinement of the former. Both define mutable-state computations over the global trace defined in Section 3.1. All trace-interacting API functions and protocol implementations are defined over one of those two effects. The following two paragraphs introduce the two layers of DY*, the unlabeled and labeled layer, and explain their relation to the effect definitions.

Unlabeled Layer and CRPYTO. The unlabeled layer is the lower layer of DY* and sets up the interaction with the trace and the attacker (e.g., through sending messages and state compromise). For trace interaction, this layer defines the effect `CRYPTO` and its Hoare-style alias `Crypto` as a mutable-state monad, including error handling. This means computations in this effect can access the trace, analyze its entries, and append new ones to it. This layer includes symbolic and concrete implementations of cryptographic operations and it is possible to verify protocols in this layer to check for guarantees with respect to the attacker. However, there are no validations of secrecy labels or trace validity.

Labeled Layer and LCRPYTO. Built on top of the unlabeled layer is the labeled one. It maintains the same guarantees as the former by relying on its API but enhances the verification process with a security type system. This layer enforces the annotation of objects with information about their security level, through type refinements in function signatures (see signatures of public key encryption Listing 3.2 and send Listing 3.3), and relies on the `can_flow` predicate, as subtype relation between labels (presented in Section 3.2). Protocol designers need to correctly identify labels of generated objects, use the labeled API functions to access the trace and perform cryptographic operations with their labeled counterparts.
This layer defines the effect `LCRYPTO` (and its Hoare-style abbreviation `LCrypto`), as a refinement of `CRYPTO`. The new effect preserves the `valid_trace` property as part of its pre- and postcondition annotations, hence labeled trace computations ensure that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

if the initial trace `t0` is valid, then calling the function leads to a new valid trace `t1`,

thereby implicitly verifying trace-invariants, as they are part of the `valid_trace` definition (Listing 3.4). As a valid trace alone does not cover the validity of usage-predicates, it is necessary to use labeled versions of cryptographic operations to verify them.

Verification of protocols at this layer is done by implementing them as functions of type `LCrypto`. The protocol is defined as a series of computational steps altering the trace (e.g., effectful API functions), from which VCs are generated, based on pre- and postcondition annotations. If the VCs are valid, the series of computation steps (aka the protocol implementation) is correct with respect to the provided function specification. By defining trace invariants and usage predicates, protocols-specific properties are captured.

3.5 Witnesses and stable Predicates

The global trace allows to reason over trace-based or safety properties. As the trace tracks each protocol step, the existence of specific operations and the time of their execution are of main interest and let one reason about the protocol flow. For example, authentication properties usually check that prior to reaching an authenticated state an associated event has been triggered before (e.g., *if both principals are not compromised then event `E` happened before event `E'`.*). It is possible to transform certain trace predicates into proven facts upon their validation, given that they hold in all succeeding states, to simplify verification and avoid a trace variable from being passed around. Thereby, it is possible to decouple the predicate from the trace and use it to do extrinsic proofs in the form of standalone lemmas in a concise way.
DY* defines a predicate to query if a specific trace entry `e` has been added at time `i`:

```
let trace_entry_at (i:timestamp)(e:entry_t):Type0= witnessed (trace_entry_at_pred i e)
```

Its definition introduces a logical construction `witnessed` (built into F*) that turns a trace-dependent predicate into a trace-independent one (see also [AFH+17]). Note that `trace_entry_at i e` is a proposition of type `Type0` and does not take a trace as an argument. Its semantics is encoded into the predicate `trace_entry_at_pred i e` of type `trace_pred` (see Listing 3.6 Line 1), which receives a trace and, therefore, can verify if the entry at index `i` is indeed `e`.

The transformation of the predicate is implemented using so-called *witnesses* over stable predicates of monotonic states, whose underlying logic has been formally defined and proven sound by Ahman et al. [AFH+17]. They identify a state definition to be *monotonic* if it grows from some state `s` to `s'` with respect to a *preorder* relation $\preceq$, i.e.:

$$\forall s \rightarrow^* s' \implies s \preceq s' \qquad (3.1)$$

where $s \rightarrow^* s'$ is true if s' is a successor state of s. If such an order exists, a stable predicate can be defined as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 type trace_pred = trace → Type0
2
3 val witness (pred:trace_pred) : CRYPTO unit
4     (λ  post s → pred s ∧ stable pred ∧ (witnessed pred  ⟹ post (Success ()) s))
5 let witness pred = assume(witnessed pred)
```

Listing 3.6: Definition of the `witness` predicate for trace reasoning

Definition (Stable Predicates) *Let ≤ be a preorder relation defined over a monotonically growing state space S then a predicate p is stable if it is valid in any successor state s' given it has been* witnessed *before in a predecessor state s:*

$$\forall s, s' : (p \ \ s) \wedge s \leq s' \implies p \ \ s'.$$

As the global trace is an append-only list, we use the *prefix* partial order (written ≤). The predicate stating that an entry `e` exists at an index `i` of the trace can be proven to be stable over ≤. That is, if an entry `e` is added at some position i to the trace, it will never disappear from that position as the trace continues to grow (or simply: *"Once something happened, it happened"*). In practice, we call a function `witness`, shown in (Listing 3.6), to propagate the validity of a stable predicate. It is defined over the `CRYPTO` effect and hence can analyze the trace (see Section 3.4 for details on effects). If its argument `pred` of type `trace → Type0` (Line 1) can be proven stable over ≤ and valid in the current state `s`, then by the definition of **Stable Predicates** it is safe to assume it will stay valid. The fact that the predicate has been *witnessed* in a prior state is conveyed through the logical construction `witnessed pred` that is independent of the trace. The proposition `witnessed pred` is brought into the SMT solver's context as a proven fact, through the `assume` statement. By the soundness proof given in [AFH+17] this operation is safe to do and the witness can be *recalled* in a later state `s'` to extract a proof of `pred s'`.
To prove the predicate `trace_entry_at i e`, for arbitrary `i` and `e`, F* actually checks if there exists a proof of `witnessed (trace_entry_at_pred i e)`, or in other words if it has been witnessed before.

This reasoning is sound but not complete: for instance, one can only prove the validity of a given witness proposition but can not make any judgments when there is no witness. The DY* framework asserts witnesses during user API calls of the unlabeled layer (therefore also by the labeled layer). In the unlabeled layer, the trace is accessed and analyzed. If a trace predicate is seen to be true, it is witnessed and its trace-independent version is made visible in the postcondition of the API call.

As an example, we take a look at how triggering an event causes the witnessed predicate to propagate, given the definition of `trigger_event` in Listing 3.7. The function takes as input the identifier `p` of the principal triggering the event and the specific event `ev` to trigger (see interface in Line 2). It is a function of the `Crypto` effect, annotated

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  (* interface *)
2  val trigger_event (p: principal) (ev: event): Crypto unit
3       (requires (λ  t0 → True))
4       (ensures (λ  t0 r t1 →
5                match r with
6                | Error _  → t0 == t1
7                | Success _  → trace_len t1 = trace_len t0 + 1 ∧
8                    did_event_occur_at (trace_len t0) p ev))
9
10 (* implementation *)
11 let trigger_event p ev =
12   let t0 = c_get () in                 (* get current trace *)
13   write_at_end (Event p ev);           (* append event entry *)
14   let trace_predicate =                (* constructing predicate *)
15        trace_entry_at_pred (trace_length t0) (Event p ev) in
16   witness trace_predicate;             (* witness predicate *)
17   assert (witnessed trace_predicate); (* verifies predicate has been witnessed *)
18   assert (did_event_occur_at (trace_length t0) p ev)
```

Listing 3.7: Interface and implementation of the effectful function to trigger protocol events

with `requires` and `ensures` clauses, which represent, respectively, the precondition, defined over the current trace `t0`, and the postcondition, which is a function over the trace `t0` before the function was triggered, the result of the computation `r` and the resulting trace `t1`. The function has a trivial precondition, while the postcondition ensures that if it returns successfully, the trace length increases, and the specified event `ev` occurs at the current timestamp. Note that `did_event_occur_at i p ev` in Line 8 is a shorthand for `trace_entry_at i (Event p ev)`.
The implementation of `trigger_event` in Listing 3.7 appends a new entry to the trace and witnesses it immediately. First the actual trace-dependent predicate `trace_predicate` on Line 15 is constructed, stating that `Event p ev` is the latest entry of the trace, before calling the function `witness` with it as an argument in Line 16. The predicate is proven to be stable and valid in the current trace as part of the WPT of `witness` (as the new trace entry has been appended to only two lines above, in Line 13, F* can easily prove `trace_predicate` to hold in the updated trace). Afterwards, the associated proposition `(witnessed trace_predicate)` is added to the SMT solver's scope, as a witness of the new entry, and the validity of the trace-independent predicate is asserted on Lines 16-18.

This may seem trivial, but this way information about the witnessed predicate is conveyed to the calling function. F* verifies that the signature in Listing 3.7 conforms with the implementation in Listing 3.7. This needs to be done only once.. From there on, whenever the function `trigger_event p ev` is called in a protocol implementation, its

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

postcondition is in scope and can be used to prove further assertions.

3.6 A dynamic Releasable Label

In the introduction (Chapter 1) and especially in the problem statement section (Section 1.2), we laid great emphasis on the intentional leakage of secret data. This is a key ingredient to many off-chain protocols like simple LN channels or HTLCs, as they require the disclosure of a private nonce at some point during protocol execution (see Section 2.3). The disclosed value is either a special *Revocation Key* used to revoke outdated transactions, or serves as proof of successful payment delivery (or receipt). The original DY⋆ framework, however, presented two limitations that were preventing us from defining the correct label for such situations:

- Firstly, as mentioned previously, DY⋆'s labeling system is *static*, and this clashes with the necessity of assigning a label that can change when a certain event occurs.
- Secondly, DY⋆ only supports *unintentional* leakage of data via state compromise, so it does not allow to model the *intentional* publication of a secret, as it happens in HTLC scripts.

This section briefly explains how we solved these issues, presenting a new label type for the DY⋆ framework, defined as

```
val releasable:(s:string)→ (l:inner_label)→ label,
```

which allows dynamic flow of a label associated with a string `s` to `public`, upon adding a special `Release (s:string)` entry to the trace.

3.6.1 Integration in DY⋆

The new label type is added to the definition of `can_flow` and requires some adaption of the current label mechanism.
We capture the semantics of `releasable (s:string)(l:inner_label)` by the following changes to the label system:

- A new type `inner_label` that represents `readers [ids]` and `public`, which we call *base* labels.
- A type `label` which is either an `inner_label`, `meet`/`join` of two `labels`, or `releasable` of a string `s` and an `inner_label`.
- We say a string `s` has been released at timestamp `i`, if the corresponding entry `Release s` has been added to the trace before or at `i`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- If the string `s` is not released yet, then `releasable s l` behaves as l.
- Once `s` has been released `releasable s l` behaves as public and can flow to any other label.

The decision to create an entirely new label, rather than "simply" adjusting `can_flow` relation, to account for trace entries of the form `Release s`, with existing labels has the reason, that `can_flow` operates solely on label types. Hence, it would not be possible to differentiate between distinct objects with the same label. This implies that we can only release per label, and not for a single secret, as we would like to do. By utilizing only existing labels we could have unwanted flows to `public` of objects annotated with the same label. Changing the signature of `can_flow` and letting it extract labeling information directly from objects themselves would cause a major revision of the whole DY⋆ framework, potentially creating new dependencies, and is therefore not an ideal solution.
As one can release only per label, refactoring the label type was inevitable. By having the new releasable label be indexed with strings they can be associated with specific objects. Using different strings for different objects lets us distinguish between them. On the other hand, it is possible to group several secrets and release them at once by attributing them to the same string `s`.

During the integration of the new label, we took care to retain the "preorder" features of the original system, like *transitivity* and *reflexivity*, but also that the new label adheres to the semantics of `meet` and `join`.
Still, some lemmas defining the behavior of `can_flow` had to be changed, as they were not applicable anymore. For example, before changing the labeling system, a flow to label `public` was only possible in the case of corruption, or via reflexivity. Now there is also a third option by releasing labels.

The release of a string `s` is captured by the global trace via a new trace entry

```
Release:s:string → entry_t (*s associated to released nonce *)
```

to indicating the release of `s`. The witnessing features of DY⋆ (Section 3.5) enable queries about the existence of the release entry and are used to alter the label flow accordingly.

When implementing protocols, principals can mark the release of their secrets by calling the API function `release (s:string)`, which appends the new entry to the trace. Additionally, we plan to add a usage-predicate, called `can_release`, to the usage predicates of Listing 3.5 introduced in Section 3.3, thus enabling thinner control over the release of labels.

3.6.2 A simple Protocol

To demonstrate our new label type we implemented a test protocol that simulates the intentional release of a nonce by `Alice` to `Bob`, as depicted in Figure 3.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

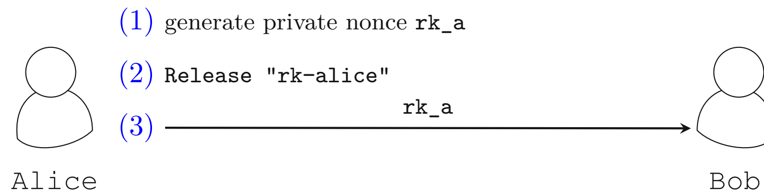


Figure 3.3: Simple protocol to demonstrate the usage of the `releasable` label type, where string `'rk-alice'` is associated with the nonce `rk_a`

The corresponding protocol implementation (`releasable_test`) is shown in Listing 3.8. It is divided into four steps:

(1) `Alice` creates a new nonce `rk_a` (stand-in for "**r**evocation **k**ey") on Line 7 and labels it with `releasable 'rk-alice' (inner_readers [P a])`, where `'rk-alice'` is the string associated with nonce `rk_a` and `inner-readers [P a]` represents the *base* label of data accessible only by principal `a` (`Alice`'s identifier). The created nonce can not flow to `public` yet (failing assertion on Line 9), but it can flow to `readers [P a]` as defined by the label semantics (succeeding assertion onLine 12).

(2) On Line 16, the label associated with the string `'rk-alice'` is released and the existence of the release entry is verified on Line 17.

(3) Now the nonce can be treated as given label `public` as asserted on Line 22.

(4) Lastly, the nonce is sent out to principal b (`Bob`'s identifier) over the network (Lines 26-27).

For now, some assertion checks still require calling explicit lemmas for verification (e.g., lines 10, 11 and 21). Future versions could further automate the verification process and delegate F* to automatically instantiate the right lemmas (through pattern matching in z3 [dMB08]), instead of having the user provide them.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
val releaseable_test (a b: principal): LCrypto unit
    (requires λ t0 → True) (ensures λ t0 idx t1 → True)

let releaseable_test a b =
    (* (1) generate revocation key *)
    let rka_label = releasable 'rk-alice' (inner_readers [P a]) in
    let (|t0, rk_a|) = rand_gen rka_label in

    (* assert(can_flow t0 rka_label public); *) (* this fails, as expected... *)
    inner_label_readableby_lemma [P a];
    releasable_can_flow_to_inner_label t0 'rk-alice' (inner_readers [P a]);
    assert(can_flow t0 rka_label (readers [P a]));

    (* (2) release nonce (this will add a Release event on the trace) *)
    let t1 = global_timestamp () in
    let t1 = release 'rk-alice' in
    assert(trace_entry_at t1 (Release 'rk-alice'));

    (* (3) check if nonce can flow to public now *)
    let t2 = global_timestamp () in
    released_label_flows_to_anything t2 'rk-alice' (inner_readers [P a]) public;
    assert(can_flow t2 rka_label public);

    (* (4) send released nonce un-encrypted *)
    let t3 = global_timestamp () in
    let now = send #t3
      a (* <- the sender *) b (* <- the receiver *) rk_a (* <- the message *)
```

Listing 3.8: Simple protocol to release a secret revocation key

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 4

Blockchain Formalization in F*

This chapter focuses on the implementation of the integrated blockchain system. First, we start with a short discussion of made assumptions and abstractions compared to the real Bitcoin blockchain (see Section 2.2). Afterwards, the section gives more detailed insights into the actual implementation within DY* and how it can be used from a user perspective. Then, we introduce an interpreter for Bitcoin scripts implemented in F* to allow (limited) guided proofs of script evaluation in the framework. At last, we define some basic blockchain properties of interest that are provable with the extension and present a simple protocol to demonstrate the added functionalities.

4.1 Abstractions and Assumptions

Trying to model every little detail of the Bitcoin blockchain is not only cumbersome but would not even be useful for our purposes, as we are interested in modeling only those details that are relevant with respect to the properties that we want to prove. For this reason, we decided to make the following abstractions:

1. First of all, we are not interested in how transactions are entered into the chain (e.g., through mining blocks and *proof-of-work*); instead we focus on the fact that they are included *eventually*, and are validated according to the consensus rules introduced in Section 2.2.1. Our model assumes a “perfect” blockchain, where the consensus algorithm is abstracted away. The model itself acts as a global instance to check if a block or transaction is valid at a given time and thereby ready to be included in the blockchain (details on block validity are discussed in Section 4.2.2). Hence, we do not formalize *proof-of-work* aspects, transaction selection, or maximizing fees/profit as these points mainly concern mining parties.

2. Assuming global consensus and keeping the blockchain in a valid state implies the existence of only one single version of the blockchain; therefore, forks are not possible.

3. Instead of creating block and transaction identifiers by hashing their contents [Eri15, Bit21], we implement them as unique random values, supplied by the blockchain system.

4. As the main objective of the model lies in off-chain protocol verification, we assume that transaction signatures do not influence the transaction identifier: the transactions follow the SegWit [Eri15] pattern. Our defined type to model transaction identifiers does not depend on the transaction data at all (see point 3 above), thus decoupling it naturally from any signatures stored in the input script[1].

5. Transactions do not include separate witness data, hence, the provided script formalization does not support *Witness Programs* [Eri15]

4.2 Blockchain Model

This section describes the integration of the blockchain model into DY*: First, we give the type definitions of the blockchain layer, relating back to the introduction of the Bitcoin blockchain in Section 2.2. Then, we show how the blockchain is embedded into the existing global trace, and finally, we touch upon the verification of correctness properties of new blocks.

4.2.1 A Definition with Types

Our goal is to keep the blockchain formalization as close to the real-world implementation as possible without unnecessarily overloading the framework. This subsection introduces the model through type definitions and explains their meaning. The application of those types and construction of the blockchain machinery is explained in Section 4.2.2. Figure 4.1 provides an overview of associated data type definitions. The notation is as follows: data types are defined as record types with their name on the gray background.

- **Blocks:** Blocks are defined via the `block_t` type and are composed of the block's own identifier (`bid`), the identifier of the previous block (`pid`), and a list of transactions (`txs`). The `pid` and `bid` are represented by unique sequences of bytes or bytestrings. A block is uniquely identified by its `bid`. The model does not specify a limit on how many transactions are placed inside a block.

- **Transactions:** Transactions follow the definition of `tx_t`. This type encompasses an identifier (`id`), a list of transaction inputs (`txin_list`), a list of transaction

[1] Reminder: PCs of the LN need to create refunding transactions (to reimburse participants in case their counterpart does not respond), that reference a funding transaction that is not signed yet. By decoupling signatures from the TXID calculation, the funding transaction can be correctly referenced.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

block_t		
bid	:	*bytes*
pid	:	*bytes*
txs	:	[*tx_t*]

txid_t		
bid	:	*bytes*
tx_idx	:	ℕ

ptr_t		
txid	:	*txid_t*
idx	:	ℕ

txout_t		
self_ptr	:	*ptr_t*
value	:	*coin*
script_pubkey	:	*script_t*

txin_t		
self_ptr	:	*ptr_t*
txout_ptr	:	*ptr_t*
value	:	*coin*
script_sig	:	*script_t*
nSequence	:	*block_height*

tx_t		
id	:	*txid_t*
txin_list	:	[*txin_t*]
txout_list	:	[*txout_t*]
nLocktime	:	*block_height*

Figure 4.1: Data types of blockchain model

outputs (txout_list) and an absolute timelock field (nLocktime). Similar to blocks, transaction input and output lists are not limited in size.

- **Inputs:** Transaction inputs, as given by txin_t, reference an (already existing) output (txout_ptr) and provide a redeeming script (script_sig) to unlock the coins held by that output. The type definition includes a self-reference (self_ptr) to ease accessing the right block and transaction during type checking. The output-reference identifies a transaction on chain and the index of the referenced output to be found in txout_list. The remaining fields refer to the amount of coins consumed (value) and the relative timelock field (nSequence).
  References to inputs (and outputs) are specified by a pointer type ptr_t, which includes the identifier of the transaction (txid) holding the input (or output) and an index (idx), to select the right input (respectively output) from the transaction's input (respectively output) list.

- **Outputs:** Transaction outputs txout_t (like inputs) carry a self-reference (self_ptr) for easier access. They contain the amount of coins that are spent (value) by that output and a locking script (script_pubkey) to set conditions for redeeming the coins.

- **TXID:** Finally, the identifier of a transaction is modeled through the type txid_t. It uniquely identifies a transaction through the identifier of the block (bid) holding the transaction, as well as its index (tx_idx) in the transaction list.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Scripts:** Input and output scripts (`script_sig` and `script_pubkey`) are represented by the type `script_t` as a list of opcodes. The type is part of our `SCRIPT` language formalization explained in Section 4.3.

The type `bytes` is built into DY⋆ and represents arbitrary bytestrings. Depending on the compilation mode, either a symbolic or concrete implementation is loaded[2]. The symbolic version includes data constructors for literals, as well as cryptographic material, like keys, and hashes.

4.2.2 Integration in DY⋆

The blockchain model is deeply integrated in DY⋆ and its trace-based reasoning. Core parts of the model are built around new trace entries, resembling blocks (via types shown in Figure 4.1) and validity checks to enforce consensus properties through the newly introduced *Blockchain Runtime Layer*. The validity checks follow the consensus rules listed in Section 2.2.1.
Besides the properties found in a consensual blockchain, like *no-double-spends*, block validity checks of our module depend on three main parts:

1. **Well-Formedness Properties**
   Due to the layered approach of DY⋆, splitting the framework into an unlabeled part and a labeled part on top, additional checks have to be performed to ensure blockchain types contain only public data.

2. **Timelock Checks**
   Absolute and relative time dependencies of the Bitcoin protocol are verified through new witnessed predicate definitions, as introduced in Section 3.5.

3. **Script Evaluation**
   A monadic interpreter implementation can evaluate scripts with concretely defined values, using F⋆'s term normalization techniques.

With the global trace at hand (cf. Chapter 3), which is simply a continuously ordered list of protocol steps indexed through timestamps, a blockchain extension comes naturally. The type of trace entries `entry_t` is expanded by a new data constructor, shown in Listing 4.1.

```
Block: pid:bytes → bid:bytes → txs:[tx_t] → entry_t
```

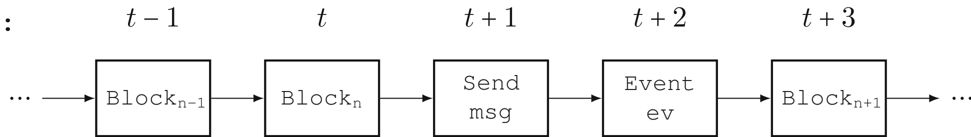
Listing 4.1: New block entry to extend global trace

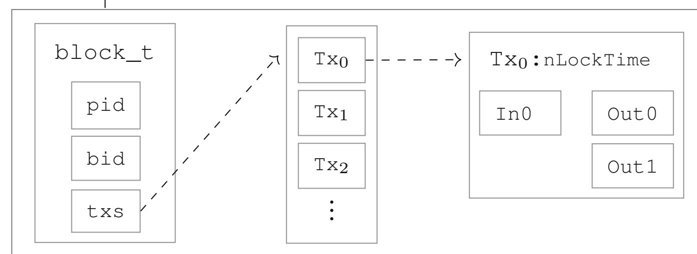
[2]Remember that DY⋆ protocols can be executed. The concrete implementation makes that execution somewhat efficient.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Blockchain:

Trace:





Legend:

- ← : indicates backward references of blocks to their predecessor.
- → : denotes the evolution of the trace (forward in time).

Figure 4.2: Global trace with block entries and extracted blockchain

The new entry represents a block, specified by the identifiers `pid` and `bid`, and its transaction list `txs`. To DY* it behaves as any other trace entry, and is therefore bound to a unique timestamp. Figure 4.2 illustrates how the blockchain is embedded in the trace, showing simplified entries. The trace includes any protocol-related execution step and each entry is associated with an index ($t-1$, t, $t+1$, ...), representing the global advance of time. Block entries can be added at any timestamp and the current blockchain can be extracted from the trace by simply ignoring non-block entries.
The figure depicts five trace entries: at timestamps $t-1$, t, and $t+3$ three block entries have been added. At timestamp $t+1$ some message `msg` was sent over the network and at $t+2$ an event `ev` was triggered. The blocks are indexed with variable n, independent of the global time t, whereas $n \leq t$ holds at any time. Below the block with index n, the underlying record type is shown to contain transaction-related information, defined by the types in Figure 4.1. Note that the index of a given block represents the absolute block height of the blockchain at the time, e.g., the index n of $Block_n$ at timestamp t, indicates the blockchain has reached height n.

Relying on the already existing global trace lets us reuse its pre-existing functionalities when it comes to trace reasoning. It also lets us interact with the witness framework

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
let was_block_mined_at j pid bid txs =
    trace_entry_at trace_index (Block pid bid txs)

let was_block_mined_before j pid bid txs =
    ∃ i. i < j ∧ was_block_mined_at i pid bid txs
```

Listing 4.2: Trace-independent predicates to verify the existence of block entries

introduced in Section 3.5 and define the predicates in Listing 4.2 describing whether a block is part of the trace. The predicate `was_block_mined_at j pid bid txs` checks for a previously witnessed trace entry of the form `Block pid bid txs` at the timestamp j, telling us if that block was "mined" at that timestamp, while the second predicate `was_block_mined_before j pid bid txs` asserts the block has been entered before j.
Recall that verification of trace-independent predicates requires a witness of the corresponding trace-dependent predicate to update the SMT solver's scopes, otherwise the proposition can not be proven correct (reasoning is not complete). Through the unlabeled backbone of DY⋆, new blocks can be mined by appending them to the trace and at the same time witnessing them.

4.2.3 Blockchain Runtime Layer

When we refer to the blockchain in DY⋆, we mean the ordered list of block entries that can be extracted from the trace. As discussed in Section 4.1 the consensus aspects are modeled through a "perfect" blockchain (i.e., one that verifies all invariants of Section 2.2.1). For that matter, the model requires each block to fulfill certain constraints before being added to the chain. To ensure a consistent view of the blockchain at all times, checks are added directly to the `valid_trace` predicate that has been introduced in Section 3.3 (remember: trace validity is a consequence of using the `LCrypto` effect). Hence, by ensuring trace validity, the validity of the blockchain invariants follows. By having these requirements in place, consensus about the blockchain is enforced during labeled protocol verification. Furthermore, one can infer those constraints back as properties when seeing a block on chain, via `valid_trace`. This stands in contrast to the real-world Bitcoin implementation, where everyone can verify the correctness of blocks themselves. However, the trust of a block being correct increases when it has received many confirmations and is part of the longest (valid) chain. Once it is buried deep enough, one may simply assume (or infer) it satisfies the consensus rules.

Consensus Validation

We define the *Blockchain Runtime Layer* to be the global instance for validation of blockchain invariants. In a sense, this layer mimics the work done by miners, who perform correctness checks on gathered transactions. We say a block is *valid* if it conforms to the correctness rules presented in Section 2.2.1, with some adjustments to comply with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the defined blockchain data types. As discussed in Section 4.1 and Section 4.2.1 the computation of TXID and BID are not modeled, hence their definition of validity differs. We now present a nested definition of valid blocks, based on the types in Figure 4.1.

Definition 1 (Block Validity) *A block of type `block_t` is considered to be valid if*

(B.1) *its `pid` references the latest block on chain,*

(B.2) *its `bid` is unique w.r.t. the global trace,*

(B.3) *its transaction list is consistent (point (3) of Section 2.2.1), and*

(B.4) *all of its transactions `txs` are valid.*

Identifiers are implemented as random numbers, that are assumed to be unique. This is a distinction to the real Bitcoin protocol, where the BID depends on the header information and the nonce "found" as a solution to the *proof-of-work* puzzle by the miner. Since each new block references the latest block on chain forks of the blockchain are prevented.
We refer to a transaction list as *consistent* if it does not contain any two transaction inputs referencing the same output.

Definition 2 (Tx Validity) *A transaction of type `tx_t` is considered valid if*

(Tx.1) *its `id` field is set correctly,*

(Tx.2) *the **absolute** height of the blockchain is greater than its `nLockTime` field (point (7) of Section 2.2.1),*

(Tx.3) *the sum of its output spendings is less or equal to the sum of coins available in its inputs (point (5) of Section 2.2.1),*

(Tx.4) *the `self_ptr` is set correctly on all outputs, and*

(Tx.5) *all of its inputs are valid.*

The `id` field of a transaction is correct if its `bid` matches the identifier of the current block and `tx_idx` identifies the transaction in the block's transaction list.
Timelock checks reject the transaction if it cannot be proven that the blockchain has reached the demanded height.
The pointer `self_ptr` has to identify the output amongst all blocks and transactions on the chain. It is set correctly if its `tx_id` refers to the `id` field of the transaction holding the output and the index `idx` identifies the output in the transaction's output list. Its correctness is verified to be sure accessing it (in a valid context) unambiguously references the output itself and also its parent transaction and block.

Definition 3 (Tx Input Validity) *A transaction input of type `txin_t` is considered valid if*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
let valid_trace (tr:trace) =
  ⋮
  ∧ (∀ (i:timestamp) (pid:bytes) (bid:bytes) (txs: tx_list_t).
    i < trace_len tr ⟹ (was_block_mined_at i pid bid txs ⟹
    is_block_valid_at i pid bid txs))
```

Listing 4.3: Extended `valid_trace` predicate with block validity checks

```
let valid_blockchain_at (j:timestamp) =
  (∀ pid bid txs i. i < j ∧ was_block_mined_at i pid bid txs ⟹
    is_block_valid_at i pid bid txs)
```

Listing 4.4: Predicate that defines a valid blockchain

(In.1) *there exists a block containing the output referenced by* `txout_ptr`*, and*

(In.2) *the* ***relative*** *age of that block is greater than the input's* `nSequence` *field (point (7) of Section 2.2.1),*

(In.3) *the referenced output is spendable (preventing* double-spends - *point (4) of Section 2.2.1),*

(In.4) *the referenced output script* `script_pubkey` *evaluates to true on the input script* `script_sig` *(point (6) of Section 2.2.1), and*

(In.5) *its* `self_ptr` *is set correctly.*

The definition of a valid transaction input is the most interesting one. It encompasses the main conditions that need to be satisfied in order to redeem an output including script evaluation **(In.4)** and the prevention of double spends **(In.3)**.

All of the above validity checks are encoded in F* and enforced by extending the `valid_trace` predicate as shown in Listing 4.3. The listing introduces a function `is_block_valid_at i pid bid txs`, which checks the validity of the block parametrized by `pid`, `bid` and `txs`, at time i, as given by Definition 1. Thus, a valid trace guarantees the validity of each block in the blockchain.
We define a blockchain, that is all the blocks that have been mined so far, to be valid at a certain timestamp i if all blocks in the chain are valid at i. The corresponding predicate is defined in Listing 4.4.

Using the updated definition of `valid_trace` and `valid_blockchain_at`, we can prove our first blockchain property: *A valid trace guarantees a valid blockchain.* The corresponding lemma definition is shown in Listing 4.5. The proof does not need manual guidance and can be discharged automatically.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
val valid_trace_implies_valid_blockchain (t0:trace) : Lemma
    (requires valid_trace t0)
    (ensures valid_blockchain_at (trace_len t0))
```

Listing 4.5: First blockchain property: a valid trace implies a valid blockchain

Well-Formedness

In addition to the validity checks, we require that all elements of a block are *well-formed*. We then say that the block itself is *well-formed*. A block's element is *well-formed* if it contains only public values, hence, it can be published.

Well-formedness mimics one of the most important features of the blockchain, its public accessibility: every principal can scan the blockchain at any time. To preserve secrecy private, objects are not allowed to be part of the blockchain: this is mainly interesting for the creation of scripts that can include arbitrary data as part of *push* opcodes. Recall that PCs of the LN and HTLCs include pre-images of hashed secrets in scripts in some protocol steps and require the secret as proof of dishonest behavior or successful payments. With well-formedness in place alone, the protocols are not provable with the standard static type system of DY*. As presented in Section 3.6 through adjustments in the labeling system it is possible to intentionally leak sensitive information and thus include them within scripts, while not violating secrecy.

Reasoning about time with the blockchain

The timelock features of the Bitcoin protocol (see Section 2.2.3) allow postponing the validity of a transaction to a later timepoint, using absolute and relative timing constraints. We define absolute timelocks in terms of block-height (i.e., current height of blockchain) and relative timelocks as block-age (i.e., depth of a specific block). To perform validity checks the *Blockchain Runtime Layer* has to verify the current height of the blockchain, as well as the depth of individual blocks (cf. validity checks in Definitions 2 and 3).

Following the approach of stable predicates over the monotonic state (see Section 3.5) to create the trace-independent `trace_entry_at` predicate, both timelock properties have been translated to stable predicates as well:

(TL.1) `block_height_at_pred i h t`:
The **absolute** block-height of the blockchain, extracted from trace `t`, is `h` at timestamp `i`

(TL.2) `block_age_at_pred i a bid t`:
The **relative** block-age of the block identified by `bid` is `a` at timestamp `i`, given the blockchain extracted from trace `t`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
script_pubkey = [OP_PUSH <public-key>; OP_CHECKSIGVERIFY]

script_sig = [OP_PUSH <valid-sig>]
```

Listing 4.6: `Pay-2-Pub-Key` output script and valid input script

From the definitions, it is easy to see why stability holds, given that the global trace adheres to the preorder $\leq$: the trace is an append-only list, and therefore, the blockchain too can only grow. It can be verified the blockchain has height exactly `h` (i.e., `h` blocks have been mined so far) at timestamp `i`, by counting all block entries up to `i`. As for any succeeding trace, the entries up to index `i` do not change given $\leq$, hence the predicate still holds.
Similarly, having verified the age of a specific block at a timestamp `i` will hold in a later trace, as entries up to `i` do not change.

The predicates **(TL.1)** and **(TL.2)** can be turned into trace-independent ones, similar to `trace_entry_at`.

4.3 SCRIPT Formalization

Scripts are one of the most essential parts of the Bitcoin blockchain, as they control the flow of coins and allow locking payments through arbitrary conditions (with respect to the scripting language). Without the inclusion of scripts protocols such as *Payment Channels* would not be possible.

The blockchain provides certain guarantees once a confirmed transaction is seen on chain: the presence of the transaction on chain implies **(1)** all input scripts have been evaluated successfully on top of the referenced output scripts (including potential timing dependencies) and **(2)** transferred coins are locked by the specified output scripts.
Note that publishing a transaction is not enough to infer meaningful properties, as miners could still choose to reject the transaction or add it at a later time.
For verification purposes, it might be beneficial to infer properties about the output lock to phrase security goals and also prove them correct. Hence, the ability to formulate proofs over script execution would enhance the verification process.

The `script_pubkey` in Listing 4.6, often referred to as `Pay-2-Pub-Key`, for example, locks the output to be only accessible by those who can present a valid signature verifiable by the given public key. Seeing a confirmed transaction output using that script implies the principal in possession of the corresponding private signature key can redeem its coins at any time. While assuming secrecy of the private key, no other principal is able to steal the coins. Of course, trying to prove larger scripts including several possible branches is more interesting.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
type op_t =
    | OP_0                          (* 0x00 *)
    | OP_PUSH: bytes → op_t         (* 0x01 - 0x4b *)
    | OP_1 | OP_2 | OP_3 | OP_4 (* 0x51 - 0x54 *)
                                         ⋮
let script_t = list op_t
```

Listing 4.7: Type definition of opcodes and scripts

In this section, we present a formalization of a SCRIPT interpreter in F*, to evaluate scripts using normalization. Unfortunately upon completion of this thesis, we were not able to achieve the desired performance, hence proving capabilities are limited.
In Section 4.3.1 we define the type of scripts and comment on our choice towards a script formalization. Afterwards in Section 4.3.2 and Section 4.3.3 we show two implementation approaches and discuss encountered problems.

4.3.1 Script Definition and Well-Formedness

Scripts are defined as a list of opcodes as shown in Listing 4.7. The type `op_t` covers a subset of opcodes from [Bit24b], including off-chain protocol relevant timelock opcodes (`OP_CHECKLOCKTIMEVERIFY` [Pet14] and `OP_CHECKSEQUENCEVERIFY` [Mar15a]) and signature checks (`OP_CHECKSIG` [Joh16]).

The type `script_t` is used for input as well as output scripts (cf. blockchain type definition of Figure 4.1). The interpreter implements a boolean evaluation function to run the concatenation of input and output script. The definitions above include the unlabeled data type `bytes`, built-into DY*. To preserve the secrecy of data embedded in scripts, the interpreter implementations should operate only on labeled types. For that matter, we define well-formed refinements of `op_t` and `script_t`. Essentially the only "problematic" opcode is `OP_PUSH (b:bytes)`, which may contain sensitive data. For readability reasons, well-formed type definitions are omitted from listings.

As shown in Listing 4.8 an alternative approach to a script interpreter implementation, could be to model scripts as arbitrary total boolean functions in F*. In this example, scripts are defined as type `bytes → bool`, i.e., functions from arbitrary bytestrings to boolean values. The type `bytes` encodes arbitrary sequences of bytes, that are concatenated together and can be split up again using the function `split`, returning `Success (b1,b2)` if the original bytestring contains sub-bytestrings `b1` and `b2`, and `Error` otherwise. The term `hash pk`, represents the hash of the public key `pk` and `verify pk h sig` denotes signature verification using the public key `pk` the message `h` and the signature `sig`. The lemma in Line 9 over the script in Line 3 can be proven easily by F*.
Still, by allowing scripts to be represented through arbitrary functions, the semantics of the Bitcoin script evaluation are missing. Firstly, SCRIPT is limited in its expressiveness

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1  type script_t = bytes → bool
2
3  let p2pkh (h:bytes) : script_t = λ wit →
4    match (split wit) with
5    | Success (sig,pk) → h = hash pk && verify pk h sig
6    | Error _ → false
7
8  (* Lemma is proven without manual guidance *)
9  let p2pkh_lemma (h:bytes) :
10   Lemma (∀ wit. p2pkh h wit ⟺ (∃ sig pk.
11     Success (sig,pk) = split wit ∧ h = hash pk ∧ verify pk h sig)) = ()
```

Listing 4.8: `Pay-2-Pub-Key-Hash` script defined as boolean function in F⋆

(e.g., non-turing complete, no loops), hence user-defined scripts could be out of the scope of the real language, thus eliminating verification results. Secondly, the language is in itself not finished and updates are usually delivered through *soft-forks*, overwriting `NO_OP` operations. Thus, opcodes may leave the stack in a different state than one would expect (e.g., the stack stays the same even though a value is consumed). Also, bugs in the evaluation of scripts can not be fixed easily, without forcing the whole Bitcoin network to jointly update. This could lead to scripts containing subtle bugs, which would be detectable through a well-defined formalization.

The formalization approaches discussed in the next sections are defined through operational semantics. Related work: The authors of [Roy23] show how to translate the semantics of a simple imperative programming language to F⋆, for teaching purposes. [KB18] and [JD23] introduce tools for the symbolic execution of Bitcoin scripts, to analyze criteria for successful execution of output scripts.

4.3.2 The Monadic Approach

In the first instance, the interpreter is implemented in a purely monadic fashion. The defined monad covers states including exceptions. The state type includes the following data:

(1) **Execution stack** (`es`): list of `bytes` to hold data relevant for execution

(2) **Operation stack** (`os`): list of `bytes` to hold branching relevant data

(3) **Current transaction** (`tx`): to access `nLockTime` field

(4) **Index of current input** (`txin_idx`): to access `nSequence` field

(5) **The output script** (`pks`): for signature verification

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Evaluation of opcodes is done according to the specification taken from [Bit24b]. The branching mechanisms utilize a second stack (*operation stack*), to push a special stack frame, containing information about which branch is taken and the current branching depth. Information about the current transaction and input, are needed for timelock checks.
Signature checks verify the signature as part of the redeeming script `script_sig` of a transaction input. When signing a transaction all input scripts are replaced by the referenced output script, and the resulting transaction serialization is hashed and signed. Currently, only this one signature mode is supported, committing to the referenced output and to the current transactions outputs list.

Evaluation is performed as small-step relation, with each opcode being implemented as a single step. For validity checks, specified in Section 4.2.2, input and output scripts are run together on empty stacks. The evaluation function is a total boolean function that recursively performs each step of the script and returns true if no errors were raised during execution.
As it turns out F⋆ has problems verifying proofs over the monadic definition. By visually examining the generated z3 queries, it seems that the monad is translated into several higher-order function applications, which could slow down the SMT solver's performance significantly.

Normalizer of F⋆. The monadic implementation works fine when all objects it operates on are fully specified and pure syntactic evaluation is run. F⋆ includes a *normalizer* to reduce terms as far as possible (via substitution) before querying z3. By either using `assert_norm` or the metaprogramming features of F⋆ [MAD+18] (to guide proofs), concrete instantiations of scripts can be run through normalization.
We also tried to guide proofs through a mix of normalization and equality rewriting, to help F⋆, but with no success. The problem is that for z3 alone the monadic representation is too big and needs to be reduced as far as possible before issuing a query. Unfortunately, one can not easily define how far terms should be reduced, only if they are reduced at all. Hence, when applying normalization it happens that already established facts can not be rewritten, as their definition has already been reduced.

Script Benchmarks. Since we can not present proofs, a set of standard scripts (shown in Listing 4.9) has been evaluated, to benchmark the interpreter, including the standard scripts (1) `Pay-To-Pub-Key` (P2PK) and (2) `Pay-To-Pub-Key-Hash` (P2PKH), as well as a multi-signature checks (3) `Pay-To-Multi-Sig` (P2MS) - (2-of-3), and an (4) `HTLC` script [LN 24, Sea17] (no HTLC revocation output).
For more information about the meaning of opcodes refer to [Bit24b]. For script (3) we ran benchmarks for all possible combinations of keys eligible to redeem coins and also verified that the wrong combinations failed. We also checked both branches of the HTLC script (4). Benchmarks were run using `assert_norm` and could be discharged by F⋆.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
p2pk = [OP_PUSH <public-key>; OP_CHECKSIGVERIFY]

p2pkh = [OP_DUP ; OP_HASH160 ; OP_PUSH (hash160 <publick-key>) ; OP_EQUALVERIFY ;
    OP_CHECKSIGVERIFY]

p2ms = [OP_2; OP_PUSH <public-key-3>; OP_PUSH <public-key-2>;
       OP_PUSH <publick-key-1>; OP_3; OP_CHECKMULTISIGVERIFY]

htlc = [OP_IF ;
          OP_SHA256 ; OP_PUSH <sha265 R> ; OP_EQUALVERIFY ; OP_DUP ; OP_HASH160 ;
          OP_PUSH (hash160 <public-key-bob>) ;
       OP_ELSE ;
          OP_PUSH <timelock> ; OP_CHECKLOCKTIMEVERIFY ; OP_DROP ; OP_DUP ;
          OP_HASH160 ; OP_PUSH (hash160 <public-key-alice>) ;
         OP_ENDIF ;
         OP_EQUALVERIFY ;
         OP_CHECKSIGVERIFY]
```

Listing 4.9: Script definitions for benchmarking the monadic interpreter

4.3.3 The Effectful Approach

We keep the definition from the monadic approach and turn the monad into an effect. Effects in F⋆ can be described by pre- and postconditions, which are proven correct over the actual implementation. Thus, by writing postconditions for each opcode, a formal proof of the specification is obtained.

Theoretically, F⋆ enables intrinsic and extrinsic proofs of effectful computations. Intrinsic proofs are done within the effect through pre- and postconditions, while extrinsic proofs require the effect to be turned into its state-passing form and then evaluated over a concrete initial state. The switch from effect to state-passing is called *reification* [AHM+19] and is only limited possible with the latest released F⋆ version to date (v2023.09.03). As far as we found out (by searching discussion forums and issue threads, release notes, and looking at the SMT queries), the current effect type of *Layered-Effects*, has only limited support for reification. This implies that proofs using the effect interpreter formalization can (so far) not be used outside the effect environment (to the best of our knowledge). Hence, lemmas reasoning over script execution for use in the blockchain model needs to be admitted, while an intrinsic proof of the same script is possible to justify this step.

As explained in Section 3.4, effects define computations with side-effects. The effectful script interpreter `BTCInt` is defined over a state monad, with the same state definition as used for the monadic interpreter. We added annotations to the step functions of opcodes, which can be proven by F⋆ within some hundred milliseconds each without further guidance. Intrinsic proofs solely depend on available postconditions, which have to be strong

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1  type post_cond_t (#a:Type) = (s0:state) → (r:result a) → (s1:state) → Type0
2
3  (* Bind for postconditions to type continuation *)
4  let bind_post_cond #a #b (pc1: post_cond_t #a) (pc2: post_cond_t #b) :
5     post_cond_t #b = λ s0 r s1 →
6        (∃ (s':state) (r':result a). pc1 s0 r' s' ∧
7           (Success?r'  ⟹ pc2 s' r s1) ∧ (* if r' is Success check pc2*)
8           (Error?r'  ⟹ Error?r ∧ s' == s1)) (* if r' is Error stop *)
```

Listing 4.10: "Bind" operation for postcondition of recursive interpreter functions

```
1  let rec eval_post_cond (sc:script_t) : post_cond_t =
2    match sc with
3    | [] → λ s0 r s1 → s0 == s1 ∧ Success?r
4    | op::sc' → bind_post_cond (step_post_cond op) (eval_post_cond sc')
```

Listing 4.11: Recursive postcondition of evaluation function

enough. F* only uses the postconditions available at the topmost level, meaning that postconditions of nested functions are blurred out. While a step function can be given a big but definitive postcondition, the recursive evaluation function requires a recursive postcondition. To concatenate an arbitrary number of postconditions Listing 4.10 defines a "bind" operation. The type `post_cond_t #a` (Line 1) defines functions that reason over an initial state `s0`, a result `r` and the final state `s1` of an interpreter step. The type `result a` on Line 1 encodes results of computations that may fail, and is either `Success r'` for some `r'` of type `a`, to denote successful execution or `Error`.
`bind_post_cont #a #b pc1 pc2` on Line 4 takes two postconditions and returns a postcondition of the computational continuation described by `pc1` and `pc2`. It introduces an existential qualifier over a potential intermediate state `s'` and an associated result `r'`. First `pc1` is verified over the initial state and intermediate values, then one has to account for failed and successful operations: if `r'` was returned successful, then `pc2` has to hold too, otherwise we stop immediately.
The postcondition of the evaluation function is shown in Listing 4.11 using `bind_post_cond`. The type `step_post_cond (op:op_t)` (Line 4) is the postcondition of the step function.

Intrinsic Script Proofs

Regarding proofs, recursive postconditions introducing new existential quantification with each operation step, seem to overlead F* quickly. Listing 4.12 shows an intrinsic proof (starting ob Line 3) of a simple script pushing the number `1` on the stack twice and verifying their equality. The proof has a trivial precondition (Line 4) and its postcondition asserts the script will run successfully and the state does not change. The proof

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
let simple_script: script_t = [OP_1; OP_1; OP_EQUALVERIFY]

let eval_simple_script : BTCInt unit
  (λ  s0 → True)
  (λ  s0 r s1 → Success?r ∧ s0 == s1) =
  eval_ test_script
```

Listing 4.12: Simple script evaluation using intrinsic proofs within the `BTCInt` effect

```
let step_p2pk (pk:bytes) (sig:bytes) : BTCInt unit
  (λ  s0 → True)
  (λ  s0 r s1 → Success?r  ⟺
       (∃ h. h = (hash (serialize (commit_tx s0.tx s0.txin_idx s0.pks))) ∧
       verify pk h sig)) =
   (* script_sig = [OP_PUSH sig] *)
   step (OP_PUSH sig);
   (* script_pubkey = [OP_PUSH pk ; OP_CHECKSIGVERIFY]*)
   step (OP_PUSH pk);
   step (OP_CHECKSIGVERIFY)
```

Listing 4.13: Intrinsic proof of `Pay-2-Pub-Key` script within `BTCInt` effect

takes about 7 seconds to complete with an increased resource limit for z3 (`z3rlimit`) on a standard consumer laptop. When splitting up the `[OP_EQUALVERIFY]` opcode into `[OP_EQUAL; OP_VERIFY]` (which does not affect the script evaluation) the proof takes already over 14 seconds. Hence, trying to prove more complex properties of longer scripts, is not practical with this approach.
However, by stripping away the "overload" of the recursive postcondition of the evaluation function and specifying each opcode by a dedicated step, proofs have way better performance. In Listing 4.13 we specify a detailed intrinsic proof under which conditions the `Pay-2-Pub-Key` output script (Listing 4.9) is satisfied. The definition of `hash` and `verify` are the same as for Listing 4.8. The function `serialize` takes a transaction and returns its serialized form as type `bytes`. For signature checks the function `commit_tx tx txin_idx pks` in Line 4, replaces `script_sig` of the input specified by `txin_idx` in `tx` with the output script `pks`. The hash of the newly obtained transaction *commits* to input to the output script. Signature checks verify against the hash of the serialized *committed* transaction. The proof takes under 10 seconds and does not require a higher resource limit but is run with the option `"split_queries always"` to split up the SMT query (see the F* tutorial [Nik24]).

Listing 4.14 shows how to verify conditions of the initial state given an "unfinished" script (e.g., only the output script). In this particular case, one can prove that the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
1 let step_p2pk' (pk:bytes) : BTCInt unit
2   (λ  s0 → True)
3   (λ  s0 r s1 → Success?r  ⟺
4        (∃ h es' sig. s0.es == sig::es' ∧
5        h = (hash (serialize (commit_tx s0.tx s0.txin_idx s0.pks))) ∧
6        verify pk h sig)) =
7    (* script_pubkey = [OP_PUSH pk ; OP_CHECKSIGVERIFY]*)
8    step (OP_PUSH pk);
9    step (OP_CHECKSIGVERIFY)
```

Listing 4.14: Intrinsic proof verifying initial stack conditions for evaluating `Pay-2-Pub-Key` script

execution stack of `s0` (`s0.es`) needs to have a signature as the top element, that is verifiable with the specific public key `pk` (from Line 4). The proof takes about 20 seconds but needs to be split and requires a higher resource limit.

Until the problems of the script evaluation are fixed or a different solution is found, protocol verification needs admitted proofs for scripts, that define the conditions for successful evaluation.

4.4 Blockchain Protocol Verification

This section provides information on how the blockchain is scanned to assert information like the uniqueness of identifiers or whether an output on chain is spendable or not. Furthermore, some blockchain properties of interest (consensus rules) of the integrated system are presented which were proven correct in F*, Lastly, we present a simple blockchain protocol implementation of `Alice` mining a new block to redeem coins from a given transaction output on chain.

4.4.1 Scanning the whole Trace

We extend the idea of witnessing single trace entries and allow trace queries over the whole trace, up to its current length. Hence, we are able to create propositions quantifying over all trace entries (i.e. `(∀ e:entry_t. P)` or `~(∃ e:entry_t. P)`, where `P` is some predicate over `e`). Standard security lemmas and trace-related propositions of DY* get along by proving the existence of a specific trace entry. Most proofs are interested in showing that either a session has been corrupted, an event was triggered, or a specific message was sent before. In contrast, the blockchain not only represents a collection of transactions but also imposes constraints on newly mined blocks, which need to be checked before insertion. A miner has to look through the entire blockchain, to decide that an output

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
(* Witness all entries in the trace *)
val witness_all (i:timestamp) : Crypto unit
  (requires λ t0 → i < trace_len t0)
  (ensures λ t0 r t1 → t0 == t1 ∧
    (match r with
    | Error _ → True
    | Success _ → ∀j. j <= i ⟹ (∃ e. (index t0 j == e) ∧ trace_entry_at j e)))

(* Wtiness all entries in the trace and prove f for all of them *)
val for_all_trace_entries (#i:timestamp) (f: entry_t → bool) : Crypto unit
  (requires λ t0 → i == trace_len t0)
  (ensures λ t0 r t1 → t0 == t1 ∧
    (match r with
    | Error _ → True
    | Success _ → ∀j e. j < i ∧ trace_entry_at j e ⟹ f e))
```

Listing 4.15: Witnessing a boolean function for all trace entries

has not been spent yet. On the other hand, the existence of a block that spends a specific output is enough, to declare that output spent.

We introduce two auxiliary functions embedded into the unlabeled runtime layer of DY⋆ defined by the signatures in Listing 4.15. These serve as assistance to verify postconditions quantifying over the whole trace and are meant to be called by API functions. `witness_all` (in Line 2) looks through the entire trace, and witnesses every entry up to the provided timestamp. The function `for_all_trace_entries` (Line 10), first witnesses every entry and then goes through the trace again to check an arbitrary (decidable) proposition `f` over trace entries. If it can prove `f` to hold for all entries then it returns successful and asserts this in its postcondition.

To check if a given transaction output `txout` is "spendable" (i.e., the output has not been referenced before by an input), every entry has to be visited, which requires *forall* quantification. Listing 4.16 defines the respective predicate that checks that no input `txin` on chain references `txout` (by comparing the output-reference of `txin` with the self pointer of `txout`), which can be verified by the method introduced above. The associated function `f: entry_t → bool` has to check for every block entry if there exists an input spending `txout`.

4.4.2 Blockchain Lemmas

The blockchain model enforces consensus rules defined in Section 2.2.1, to establish a valid version of the blockchain. Following those rules Section 4.2 describes a notion of *block validity* through Definitions 1, 2 and 3 given the internal type representations in F⋆. By adding validity checks to the definition of a valid trace and by requiring protocol

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
(* ~~~ Check if txout has already been spend (double spending) ~~~ *)
let is_txout_spendable_at (i: timestamp) (txout: txout_t) =
  (∀ (txin: txin_t). exist_block_with_txin_before i txin ⟹
    ~(txin.txout_ptr == txout.self_ptr))
```

Listing 4.16: Verify a given transaction output `txout` is "spendable"

implementations to maintain a valid trace throughout execution, consensus rules are followed. In Listing 4.5 we present a lemma to prove that *"a valid trace implies a valid blockchain"*.
To verify the blockchain implementation behaves as expected and to support proofs of security goals, consensus rules are formulated as lemmas and proven correct in $F^\star$. Below we define a set of lemmas (in textual description) that cover properties of interest, most noteworthy is Lemma 4, which states that no two distinct inputs can reference a single output.

Lemma 1 (Unique BID) *If there are two blocks with the same BID on a valid blockchain, then those blocks are the same.*

Lemma 2 (Transaction Spending) *Each transaction that is part of a valid blockchain spends less coins than the sum of its inputs.*

Lemma 3 (Reference Spenadable TxOut) *Each transaction input that is part of a valid blockchain references only transaction outputs, that have not been spent before.*

Lemma 4 (No Double-Spends) *A valid blockchain does not contain any two different transaction inputs, that reference the same transaction output.*

Lemma 5 (Blockchain Secrecy) *Each block that is part of a valid blockchain only contains public data.*

Lemma 6 (TxOut Spending Conditions) *A transaction input `txin` that is part of a valid blockchain, can spend a referenced transaction output `txout` only if the following conditions are met:*

- *The referenced `txout` exists and is part of the chain before `txin`.*
- *The blockchain has reached the absolute block-height specified by the nLockTime field of the transaction containing `txin`.*
- *The block containing `txout` has reached the depth (or age) specified by the nSequence field of `txin`.*
- *The output script `script_pubkey` of `txout` evaluates successful on the the input script `script_sig` of `txin`.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

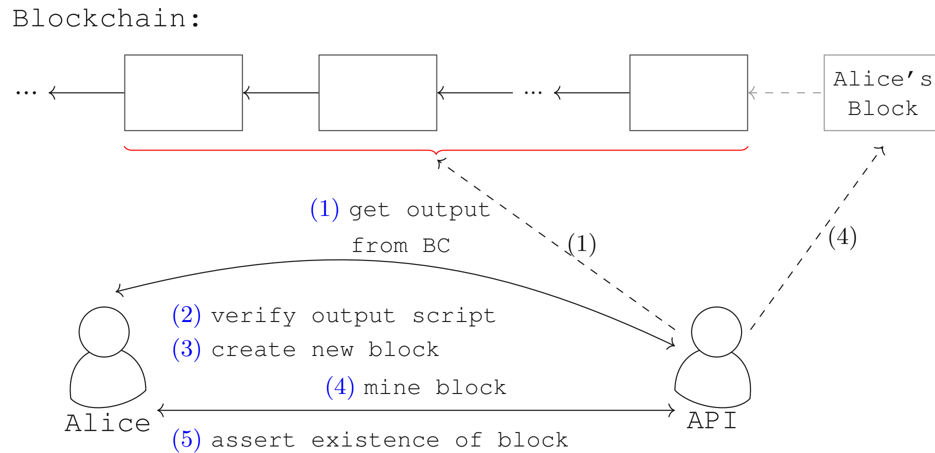


Figure 4.3: A simple blockchain protocol to redeem a given output on-chain

Some lemmas need guidance through auxiliary lemmas, but mostly no additional compiler options for increased resources or fuel are needed. The blockchain machinery including validity lemmas is defined in the `BlockChainRuntime` module, which is verified in about 30 seconds. The verification has to be done only once after downloading the DY* code, afterwards, the proven lemmas can be used for further proof support.

4.4.3 A simple Blockchain protocol

To demonstrate the functionalities of the blockchain system we implemented a simple protocol as illustrated in Figure 4.3. `Alice` analyzes the blockchain through the API (see appendix for API signatures) and mines a new block to redeem coins locked in an output on-chain.

At the beginning `Alice` gets a pointer to the output (including the identifier of the block containing the output and the output's index), she then performs step (1) and checks the blockchain for the given output. If the output can be found on-chain `Alice` verifies the output script (step (2)) and creates a corresponding input script. Then in step (3) she creates a new block, including the transaction to spend the output, and mines it in step (4). The mining process verifies the validity of the block, thereby preserving a valid blockchain. If the block was mined successfully `Alice` can verify its existence through the postcondition of the mining function (i.e., the new block has been witnessed - step (5)). Additionally, `Alice` can confirm the referenced output is not spendable anymore.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Improvements and Future Work

In this work, we showed how a Bitcoin blockchain layer can be encapsulated into the DY* framework for symbolic protocol verification. Thereby we could create a basis for future blockchain-related verification techniques, especially for off-chain protocols.
To reach this goal we have identified the following key points and possible improvements to be the subject of future work:

- Support F* during verification with additional lemmas about block validity checks, to prevent performance issues due to the nested record types.
- Merging the existing blockchain layer with the new labeling system to release secret data and create a stable version.
- Create protocol abstractions and define security goals for off-chain protocols such as PC and HTLCs.
- Formalize the protocols step by step in DY* together with the targeted properties, and the verification techniques shown in this thesis.
- Improve the scripting part of the blockchain layer, by either enabling proof assistance of the current scripting system or resorting to a different approach, which could be
    1. switching away from the monadic implementation to a pure state passing one, to eliminate higher-order function applications in the SMT encoding,
    2. down-scaling the language to a simpler imperative grammar,
    3. using total boolean functions over bytestrings, or
    4. revise the inclusion of F*'s effect system (if possible) with a combination of intrinsic and extrinsic proofs using reification.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Conclusion

This thesis focused on establishing a basis for verifying off-chain protocol implementations, specifically targeting the Bitcoin protocol and the *Lightning Network*. The presented approach utilizes type system based reasoning and proof-assistance using the symbolic verification framework DY*, written in the proof-oriented programming language F*.

In the introduction, we highlighted two essential aspects typical to the mentioned off-chain protocols: (1) the intentional disclosure of secret data, and (2) the inclusion of timeout periods for issuing refunds or punishing malicious actions. To facilitate the verification of off-chain protocols, the methods utilized should support both of these characteristics.

We extended the symbolic verification framework DY* by adapting its labeling system and the global trace infrastructure to support blockchain-like data types. A new label type `releasable (l:inner-label)(s:string)` formalizes the ability to *release* the annotated data object to type `public` once a corresponding release entry, associated to string `s`, is entered to the trace. Furthermore, a blockchain system is integrated, based on new global trace entries, resembling Bitcoin blocks. Bitcoin is built around a consensus protocol to keep the blockchain in a valid state all the time. Our system guarantees the inclusion of valid blocks only, thus mimicking the consensus mechanisms.

Validity checks rely on three main components:

1. **Well-formedness**: Given the blockchain's public accessibility, each block must exclusively contain public data to preserve overall secrecy.

2. **Timelocks**: The blockchain can be extracted from the trace by filtering out all non-block entries. Specialized *trace-predicates* are utilized to verify timing constraints related to both absolute *block-height* and relative *block-age*.

3. **Script evaluation**: The formalization of SCRIPT in F* enables the evaluation of semantically fully specified scripts through normalization and provides limited

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

proof capabilities. A small series of benchmarks has been conducted to assess the implementation and build confidence in its accuracy.

We demonstrated the adherence of the implemented blockchain model to a set of consensus rules derived from the Bitcoin protocol by formalizing them as lemmas and verifying their correctness. Additionally, to showcase the usability of our extension, we include implementations of two simple protocols, one for the labeling and one for the blockchain part.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

1.1 Illustration of a LN *Payment Channel* 4

2.1 Illustration of two Bitcoin transactions 19
2.2 Stack evaluation of `Pay-To-Pub-Key-Hash` script on valid input script . 20
2.3 Example transaction to show notation used in this section 22
2.4 PC Commitment Transaction created by `Alice` 24
2.5 PC Commitment Transaction created by `Bob` 25
2.6 Successful Multi-Hop payment with HTLCs 27
2.7 Commitment Transaction with HTLC output created by `Alice` 28
2.8 Commitment Transaction with HTLC output created by `Bob` 28
2.9 Simplified illustration of information leakage of Multi-Hop payments with HTLCs 30

3.1 Evolution of the global trace after triggering an event 33
3.2 Lattice order of *can-flow-relation* with corruption 35
3.3 Simple protocol to demonstrate the usage of the `releasable` label type, where string `'rk-alice'` is associated with the nonce `rk_a` 45

4.1 Data types of blockchain model 49
4.2 Global trace with block entries and extracted blockchain 51
4.3 A simple blockchain protocol to redeem a given output on-chain 66

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Listings

2.1 Pay-2-Pub-Key-Hash script 20
3.1 The type entry_t of global trace entries 32
3.2 Simplified signature of the public-key encryption function pke_enc 36
3.3 Signature of send function with label information on message 37
3.4 Definition of a valid trace 37
3.5 Type of usage-predicates 38
3.6 Definition of the witness predicate for trace reasoning 41
3.7 Interface and implementation of the effectful function to trigger protocol events 42
3.8 Simple protocol to release a secret revocation key 46
4.1 New block entry to extend global trace 50
4.2 Trace-independent predicates to verify the existence of block entries . . 52
4.3 Extended valid_trace predicate with block validity checks 54
4.4 Predicate that defines a valid blockchain 54
4.5 First blockchain property: a valid trace implies a valid blockchain . . . 55
4.6 Pay-2-Pub-Key output script and valid input script 56
4.7 Type definition of opcodes and scripts 57
4.8 Pay-2-Pub-Key-Hash script defined as boolean function in F* 58
4.9 Script definitions for benchmarking the monadic interpreter 60
4.10 “Bind” operation for postcondition of recursive interpreter functions . . 61
4.11 Recursive postcondition of evaluation function 61
4.12 Simple script evaluation using intrinsic proofs within the BTCInt effect 62
4.13 Intrinsic proof of Pay-2-Pub-Key script within BTCInt effect 62
4.14 Intrinsic proof verifying initial stack conditions for evaluating Pay-2-Pub-Key script 63
4.15 Witnessing a boolean function for all trace entries 64
4.16 Verify a given transaction output txout is “spendable” 65

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[ABLZ18] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security*, pages 541–560, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg.

[AFH+17] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[AHM+19] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martinez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. `https://arxiv.org/abs/1608.06499`, 2019.

[BBD+21a] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. *A Tutorial-Style Introduction to DY⋆*, pages 77–97. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Science and Business Media Deutschland GmbH, Germany, 2021.

[BBD+21b] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY⋆: A modular symbolic verification framework for executable cryptographic protocol code. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542, 2021.

[BBF+08] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32, 2008.

[BBKM23] Stefano Bistarelli, Andrea Bracciali, Rick Klomp, and Ivan Mercanti. Towards automated verification of bitcoin-based decentralised applications. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 262–269, New York, NY, USA, 2023. Association for Computing Machinery.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[BCL12] Gergei Bana and Hubert Comon-Lundh. Towards unconditional soundness: Computationally complete symbolic attacker. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 189–208, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BDJ+21] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 537–554, 2021.

[BDS15] David Basin, Jannik Dreier, and Ralf Sasse. Automated symbolic proofs of observational equivalence. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1144–1155, New York, NY, USA, 2015. Association for Computing Machinery.

[BFG10] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 445–456, New York, NY, USA, 2010. Association for Computing Machinery.

[BFGT06] K. Bhargavan, C. Fournet, A.D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 14 pp.–152, 2006.

[BGW20] Colin Boyd, Kristian Gjøsteen, and Shuang Wu. A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract. In Bruno Bernardo and Diego Marmsoler, editors, *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, volume 84 of *OpenAccess Series in Informatics (OASIcs)*, pages 5:1–5:13, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[BHM14] Michael Backes, Cătălin Hriţcu, and Matteo Maffei. Union, intersection, and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security*, 22:301–353, 03 2014.

[Bit21] Bitcoin Community. Block hashing algorithm. `https://en.bitcoin.it/wiki/Block_hashing_algorithm`, 2021. [Online; accessed 30-April-2024].

[Bit24a] Bitcoin Community. Bitcoin improvement proposals. `https://github.com/bitcoin/bips`, 2024. [Online; accessed 30-April-2024].

[Bit24b] Bitcoin Community. Script. `https://en.bitcoin.it/wiki/Script`, 2024. [Online; accessed 3-April-2024].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Bla06] B. Blanchet. A computationally sound mechanized prover for security protocols. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15 pp.–154, 2006.

[Bla12] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust*, pages 3–29, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Bla14] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: the verifier ProVerif. In Alessandro Aldini, Javier Lopez, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2014.

[Blo24] Blockchain.com. Average transactions per block. `https://www.blockchain.com/explorer/charts/n-transactions-per-block`, 2024. [Online; accessed 7-April-2024].

[BMV05] David Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, Jun 2005.

[BZ19] Massimo Bartoletti and Roberto Zunino. Formal models of bitcoin contracts: A survey. *Frontiers in Blockchain*, 2, 2019.

[Can01] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[CFvdPS15] Kaylash Chaudhary, Ansgar Fehnker, Jaco van de Pol, and Mariëlle Stoelinga. Modeling and verification of the bitcoin protocol. In *MARS*, 2015.

[CGLM17] Véronique Cortier, Niklas Grimm, Joseph Lallemand, and Matteo Maffei. A type system for privacy properties. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 409–423, New York, NY, USA, 2017. Association for Computing Machinery.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *2008 Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340. Springer, Berlin, Heidelberg, March 2008.

[DOA16] Richard Dennis, Gareth Owenson, and Benjamin Aziz. A temporal blockchain: A formal analysis. In *2016 International Conference on Collaboration Technologies and Systems (CTS)*, pages 430–437, 2016.

[DSST+22] Saulo Dos Santos, Japjeet Singh, Ruppa K. Thulasiram, Shahin Kamali, Louis Sirico, and Lisa Loud. A new era of blockchain-powered decentralized finance (defi) - a review. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1286–1292, 2022.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[Eri15] Eric Lombrozo and Johnson Lau and Pieter Wuille. Segregated witness (consensus layer). `https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki`, 2015. [Online; accessed 18-July-2023].

[FM11] Riccardo Focardi and Matteo Maffei. Types for security protocols. In Véronique Cortier and Steve Kremer, editors, *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 143–181. IOS Press, 2011.

[Gav11] Gavin Andresen. M-of-n standard transactions. `https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki`, 2011. [Online; accessed 30-April-2024].

[Gav13] Gavin Andresen and Mike Hearn. Payment protocol. `https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki`, 2013. [Online; accessed 7-October-2023].

[GH20] Matthias Grundmann and Hannes Hartenstein. Fundamental properties of the layer below a payment channel network (extended version). `https://arxiv.org/abs/2010.08316`, 2020.

[GH22] Matthias Grundmann and Hannes Hartenstein. Verifying payment channels with tla+. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3, 2022.

[GH23] Matthias Grundmann and Hannes Hartenstein. Towards a formal verification of the lightning network with tla+. `https://arxiv.org/abs/2307.02342`, 2023.

[GMAA22] Ahmed G. Gad, Diana T. Mosa, Laith Abualigah, and Amr A. Abohany. Emerging trends in blockchain technology and applications: A review and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

outlook. *Journal of King Saud University - Computer and Information Sciences*, 34(9):6719–6742, 2022.

[GMSR+19] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. Cryptology ePrint Archive, Paper 2019/360, 2019. `https://eprint.iacr.org/2019/360`.

[JD23] Vincent Jacquot and Benoît Donnet. Chaussette: A symbolic verification of bitcoin scripts. In *Proc. International Workshop on Cryptocurrencies and Blockchain Technology (CBT)*. Springer, September 2023.

[Joh16] Johnson Lau and Pieter Wuille. Transaction signature verification for version 0 witness program. `https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki`, 2016. [Online; accessed 30-April-2024].

[KB18] Rick Klomp and Andrea Bracciali. On symbolic verification of bitcoin's script language. In *DPM/CBT@ESORICS*, 2018.

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450, 2017.

[KK18] Nadim Kobeissi and Natalia Kulatova. Ledger design language: Towards formal reasoning and implementation for public ledgers. Cryptology ePrint Archive, Paper 2018/416, 2018. `https://eprint.iacr.org/2018/416`.

[KL20] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 334–349, 2020.

[LN 16] LN Community. Bolt: Basis of lightning technology (lightning network specifications). `https://github.com/lightning/bolts/`, 2016. [Online; accessed 3-November-2023].

[LN 24] LN Community. Bitcoin transaction and script formats. `https://github.com/lightning/bolts/blob/master/03-transactions.md`, 2024. [Online; accessed 30-April-2024].

[LNB+15] Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and P. Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. *IACR Cryptol. ePrint Arch.*, 2015:1168, 2015.

[Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[LV20] Cosimo Laneve and Adele Veschetti. A Formal Analysis of the Bitcoin Protocol. In Frank S. de Boer and Jacopo Mauro, editors, *Recent Developments in the Design and Implementation of Programming Languages*, volume 86 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[MAD+18] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-f*: Proof automation with smt, tactics, and metaprograms. Technical Report MSR-TR-2018-33, Microsoft, July 2018.

[Mar15a] Mark Friedenbach and BtcDrak and Eric Lombrozo. Checksequenceverify. `https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki`, 2015. [Online; accessed 24-April-2024].

[Mar15b] Mark Friedenbach and BtcDrak and Nicolas Dorier and kinoshitajona. Relative lock-time using consensus-enforced sequence numbers. `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki`, 2015. [Online; accessed 24-April-2024].

[Maz22] Subhra Mazumdar. Towards faster settlement in htlc-based cross-chain atomic swaps. `https://arxiv.org/abs/2211.15804`, 2022.

[MDPM18] Daniela Mechkaroska, Vesna Dimitrova, and Aleksandra Popovska-Mitrovikj. Analysis of the possibilities for improvement of blockchain technology. In *2018 26th Telecommunications Forum (TELFOR)*, pages 1–4, 2018.

[MMSH16] Patrick McCorry, Malte Möser, Siamak F. Shahandasti, and Feng Hao. Towards bitcoin payment networks. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy*, pages 57–76, Cham, 2016. Springer International Publishing.

[MMSK+17] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 455–471, New York, NY, USA, 2017. Association for Computing Machinery.

[MMSS+19] Giulio Malavolta, Pedro A. Moreno-Sánchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous multi-hop locks for blockchain scalability and interoperability. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[MPJ+16] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of bitcoins:

Characterizing payments among men with no names. *Commun. ACM*, 59(4):86–93, mar 2016.

[MPSW19] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Designs, Codes and Cryptography*, 87:2139 – 2164, 2019.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[MSMH21] Paolo Modesti, Siamak F. Shahandashti, Patrick McCorry, and Feng Hao. Formal modelling and security analysis of bitcoin's payment protocol. *Computers & Security*, 107:102279, 2021.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, 2009.

[ND18] Neha Narula and Tadge Dryia. MAS.S62, Cryptocurrency Engineering And Design, Spring 2018, Lecture 13: Payment Channels and Lightning Network. `https://ocw.mit.edu/courses/mas-s62-cryptocurrency-engineering-and-design-spring-2018/`, 2018. (MIT OpenCourseWare: Massachusetts Institute of Technology). [Online; accessed 30-April-2024]. License: Creative commons BY-NC-SA.

[Nik24] Nikhil Swamy and Guido Martínez, and Aseem Rastogi. Proof-oriented programming in f*. `https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf`, 2024. [Online; accessed 19-April-2024].

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21:993–999, 1978.

[PD16] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. `https://lightning.network/lightning-network-paper.pdf`, 2016.

[Pet14] Peter Todd. Op_checklocktimeverify. `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`, 2014. [Online; accessed 24-April-2024].

[Pie20] Pieter Wuille and Jonas Nick and Tim Ruffing. Schnorr signatures for secp256k1. `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`, 2020. [Online; accessed 8-October-2023].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Roy23] Subhajit Roy. A theorem proving approach to programming language semantics. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '23, page 153–165. IEEE Press, 2023.

[SCL13] Nikhil Swamy, Juan Chen, and Ben Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM Programming Language Design and Implementation (PLDI) 2013*. ACM, June 2013.

[Sea17] Sean Bowe and Daira Hopwood. Hashed time-locked contract transactions. `https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki`, 2017. [Online; accessed 24-July-2023].

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F⋆. In *POPL '16 Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, January 2016.

[SMCB12] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of diffie-hellman protocols and advanced security properties. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 78–94, 2012.

[Sta22] Statista. Fee per bitcoin transaction from october 2006 to november 14, 2022. `https://www.statista.com/statistics/731459/bitcoin-transaction-fees/`, 2022. [Online; accessed 14-July-2023].

[Sta23] Statista. Bitcoin (btc) price per day from apr 2013 - jul 10, 2023. `https://www.statista.com/statistics/326707/bitcoin-price-index/`, 2023. [Online; accessed 7-April-2024].

[Vis23] Visa. Annual report 2023. `https://annualreport.visa.com/financials/default.aspx`, 2023. [Online; accessed 19-April-2024].

[WSNH19] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. Cryptology ePrint Archive, Paper 2019/1178, 2019. `https://eprint.iacr.org/2019/1178`.

[YXL+22] Jiangyuan Yao, Chunxiang Xu, Deshun Li, Shengjun Lin, and Xingcan Cao. Formal verification of security protocols: Proverif and extensions. In *Artificial Intelligence and Security: 8th International Conference, ICAIS 2022, Qinghai, China, July 15–20, 2022, Proceedings, Part II*, page 500–512, Berlin, Heidelberg, 2022. Springer-Verlag.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Appendix

Blockchain API

This section shows the implemented API function for blockchain access. Listing 1 defines the function signature to mine new blocks. It returns the current height of the blockchain (after adding the new block). The precondition checks for block validity (Line 4), to keep the trace valid and includes a user-defined trace invariant `mine_pred_at` to control the mining of blocks. Its postcondition ensures the new black was mined at the current timestamp (Line 6) and the blockchain reached the absolute block height that has been returned (Line 7).

In Listing 2 shows a function signature to determine the block height at a given timestamp for additional checks. The function ensures that the state does not change and the returned height has been reached at the specified time (Line 4). As explained in Section 4.2.3 verifying timelock specification is done through witness predicates.

```
val mine_block: #i:timestamp → pid:bytes → bid:bytes →
        txs: tx_list_t → LCrypto block_height
    (requires (λ  t0 → i == trace_len t0 ∧ mine_pred_at i pid bid txs ∧
       is_block_valid_at i pid bid txs)
    (ensures (λ  t0 h t1 → trace_len t1 == trace_len t0 + 1 ∧
        was_block_mined_at (trace_len t0) pid bid txs ∧
        trace_block_height_reached_at (trace_len t1) h))
```

Listing 1: User API function to mine new blocks

```
val get_block_height: i:timestamp → LCrypto block_height
    (requires (λ  t0 → i <= trace_len t0))
    (ensures (λ  t0 h t1 → t0 == t1 ∧ i <= trace_len t1 ∧
       trace_block_height_reached_at i h))
```

Listing 2: User API function to determine the current height of the blockchain

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
val get_txout : block_index:timestamp → txout_ptr:ptr_t →
   LCrypto (now:timestamp & txout:txout_t & block_age:block_height)
   (requires (λ  t0 → True))
   (ensures (λ  t0 (|now,txout,block_age|) t1 → t0 == t1 ∧
      now == trace_len t0 ∧ block_index < now ∧
      is_well_formed_txout #now txout ∧
      txout.self_ptr == txout_ptr ∧
      exist_block_with_txout_at block_index txout ∧
      trace_block_age_at now block_age txout.self_ptr.tx_id.bid))
```

Listing 3: User API function to search and return a specific transaction output

```
val verify_txout : txout:txout_t → script_pubkey:script_t →
   LCrypto unit pr
   (requires (λ  t0 → True ))
   (ensures (λ  t0 (s) t1 → t0 == t1 ∧ txout.script_pubkey == script_pubkey ∧
      is_txout_spendable_at (trace_len t0) txout))
```

Listing 4: User API function to verify properties of given transaction output

To spend specific transaction outputs on the chain, the given output has to be validated first, to ensure it contains the right script and is has not been spent before. An output can be extracted from the global trace by calling the function in Listing 3. It gets the index of the block containing the output and the corresponding pointer and returns the target output and its current age. The postcondition ensures the output is well-formed (Line 6), as can be inferred from its presence on the chain, its self-reference equals the specified output pointer (Line 7), its is part of the block mined at the given index (Line 8) and the returned age has been witnessed before (Line 9).

The extracted output is verified with the function defined in Listing 4. It gets a specific output and an anticipated output script, and ensures that the script of the provided output matches the specified output script (to build a corresponding input script - Line 4) and that the output is indeed spendable at the current timestamp (Line 5).

New block identifier are generated at random and witnessed to be unique by traversing the trace. The function signature in Listing 5 returns a fresh random bytestring and asserts its uniqueness (Line 4).

Besides mining complete blocks, we added a function to publish single transactions to the network (or to the miners), which is closer to the real Bitcoin network. Its signature is shown in Listing 6 and it takes the current timestamp, a publisher id, and the transaction to be published as arguments. The precondition requires the given transaction to be valid (i.e., preserve valid trace properties - Line 3). The postcondition asserts the transaction

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 val generate_bid : unit → LCrypto (now:timestamp & bid:msg now public)
2     (requires (λ  t0 → True))
3     (ensures (λ  t0 (|now,bid|) t1 → trace_len t1 = trace_len t0 + 1 ∧
4        now = trace_len t0 ∧ is_bid_unique_at (trace_len t1) bid))
```

Listing 5: User API function to generate a fresh *block-id*

```
1 val publish_tx: #i:timestamp → publisher:principal → tx:tx_t →
2     LCrypto timestamp
3     (requires (λ  t0 → i == trace_len t0 ∧ is_tx_valid_at i tx))
4     (ensures (λ  t0 r t1 → r == trace_len t0 ∧
5        trace_len t1 = trace_len t0 + 1 ∧
6        was_message_sent_at (trace_len t0) publisher 'MINER' (serialize tx)))
```

Listing 6: User API function to publish a transaction to the network

has been serialized and sent over the network (Line 6), which implies the trace length increased by one (Line 5). Internally this function is a wrapper for `send` (see Listing 3.3) to enter a send event to the trace with the receiver being attributed to the string `'MINER'`. As the attacker's knowledge grows with every send event the transaction is disclosed upon publishing it. Via deduction rules, the attacker can derive the transaction and its internal types, such as scripts to extract embedded data.
As publishing a transaction alone does not justify inferring its presence on the blockchain (cmp. discussion in Section 4.3), we do not learn more than that the transaction has been sent out. If reasoning requires the presence of a certain transaction on chain, it has to be validated separately (i.e., analyze whole blockchain), as one would in the real world.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN