Security Aspects of Payment Channel Networks

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Matthias Alfred Grundmann

Tag der mündlichen Prüfung:	29. Juni 2026
Erster Referent:	Prof. Dr. rer. nat. Hannes Hartenstein Karlsruher Institut für Technologie
Zweiter Referent:	Prof. Dr. rer. nat. Florian Tschorsch Technische Universität Dresden

Acknowledgements

First and foremost, I would like to express my deep gratitude to my advisor, Hannes Hartenstein, for his guidance, support, and mentorship throughout my doctoral studies. I am particularly grateful for the freedom he gave me to choose and shape my research topics, as well as for his patience and support as I familiarized myself with new areas of research. I would also like to thank Florian Tschorsch for serving as co-referee for this thesis and for his helpful comments and thought-provoking questions.

My sincere thanks go to my colleagues in the DSN research group with whom I had the pleasure of working over the years: Saskia Bayreuther, Jan Droll, Jan Grashöfer, Niklas Hemken, Florian Jacob, Marc Leinweber, Till Neudecker, Patrick Spiesberger, Oliver Stengele, and Christina Westermeyer. Thank you for the many fruitful discussions and your honest and constructive feedback.

I am grateful to all the students whose theses and research projects I supervised for their valuable contributions and the fresh perspectives they brought. In particular, I would like to mention those who co-authored papers with me: Hedwig Amberg, Max Baumstark, Hannes Bönisch, Leonard Schönborn, and Otto von Zastrow-Marcks.

I would also like to thank the many anonymous referees who reviewed the papers on which this thesis is based. Their valuable comments highlighted critical aspects that required further work and helped improve the quality of the research. At the same time, their recognition of the strengths they saw in each paper provided encouragement and motivation.

Finally, I am deeply grateful to my family for their unwavering support. In particular, I thank my wife and children for their support, patience, and encouragement throughout this journey.

Abstract

Conventional payment systems rely on trusted parties such as banks or payment service providers. Bitcoin [Nak08] introduced a decentralized approach that replaces a central trusted party by a peer-to-peer network in which consensus is established over the state of a blockchain. A blockchain is a data structure that stores all transactions and is maintained without a central party. Having access to all transactions, each user can verify whether a unit of money has been spent. However, writing to the blockchain creates a performance bottleneck because the Bitcoin peers must regularly reach consensus over the state of the blockchain. Recently, payment channel networks [PD16] have been proposed which are an approach that is rooted in a blockchain but allows for peer-to-peer payments without writing to the blockchain and, thus, is not limited by the same performance bottleneck. Lightning is an implementation of a payment channel network for Bitcoin that is increasingly being used. In this thesis, we explore security aspects of payment channel networks using the example of Lightning.

We focus on the Lightning protocol and develop an approach to mechanically check the security of Lightning. We focus on the layer below a payment channel network, show design options for this layer, and empirically analyze the peer-to-peer network of Bitcoin. We focus on the user side of payment channels and develop two approaches for users to involve third parties to secure their payment channels. Finally, we focus on applications using a payment channel network and develop a protocol for ticketing in public transport whose security properties are based on a payment channel network.

The security of the Lightning protocol has been investigated previously. In contrast to previous work, we aim for a machine-checked verification of the security of Lightning and lay the groundwork for such a verification: We contribute a formal specification of the security property and a formal specification of the Lightning protocol in the formal specification language TLA$^+$. We develop a chain of abstractions from the specification of the Lightning protocol to the security property. By combining model checking and proofs to show the correctness of these abstractions, we show that the Lightning protocol fulfills the security property. While our validation relies on finite model checking, it gives sufficient confidence that Lightning fulfills the security property so that a complete machine-checked proof can be approached in future work.

A payment channel network relies on an underlying layer. We show that this underlying layer cannot only be implemented by a blockchain but could also be implemented by a network of banks or a central bank digital currency. The Lightning Network is based upon Bitcoin as a first layer. Previous work has shown that Bitcoin fulfills the requirements that the Lightning Network has for its underlying layer based on assumptions on the

peer-to-peer network used by Bitcoin. Empirically, it can be seen that these assumptions are met by the current peer-to-peer network of Bitcoin, however, little is known about the peer-to-peer network itself. In an empirical part of this thesis, we collect and analyze evidence on previously unknown metrics about the peer-to-peer network such as the number of unreachable peers in the network or the peer degree of reachable peers. We find that in April 2026, there are about 50 000 unreachable peers in the Bitcoin peer-to-peer network. We also find that more than 50 % of the reachable peers have a peer degree that is close to the default connection limit which indicates that those peers cannot accept additional incoming connections and the network's connection capacity is close to being depleted.

The benefits of payment channel networks such as the ability to process a high number of transactions and reducing transaction latency come along with the requirement for users to regularly monitor the underlying layer for fraudulent transactions and react to such transactions within a certain time window. Approaches have been proposed that allow a user to securely transfer this task to a third party, called a watchtower. We develop two novel watchtower approaches. The first approach makes watchtower approaches economically practical by using trusted execution environments to reduce the storage requirements of a watchtower provider. The second watchtower approach removes the requirement of direct interaction between a user and a watchtower. Thereby, a user's dependence on a specific watchtower is reduced because a payment channel can be protected by any watchtower.

Finally, we show the practical relevance of payment channel networks by applying them in the context of public transport ticketing. A common setting for public transport is that there are multiple providers operating a public transport network and travelers can buy a ticket from one provider and use the ticket for routes operated by a different provider. We propose a protocol that uses a payment channel network for ticket payment. By using a payment channel network, the protocol achieves a fair exchange of a ticket and the payment for a ticket between a customer and a provider and each payment is directly sent to the provider operating the vehicles used by the customer.

In conclusion, this thesis demonstrates multifaceted security aspects of payment channel networks and contributes approaches to make payment channel networks and their usage more secure. Thereby, this thesis is a step towards future payment systems that are practical, decentralized, and scalable.

Zusammenfassung

Konventionelle Zahlungssysteme basieren auf vertrauenswürdigen Parteien wie Banken oder Zahlungsdienstleistern. Mit Bitcoin [Nak08] wurde ein dezentraler Ansatz eingeführt, bei dem an die Stelle einer zentralen Vertrauensinstanz ein Peer-to-Peer-Netzwerk tritt, in dem Konsens über den Zustand einer Blockchain hergestellt wird. Eine Blockchain ist eine Datenstruktur, die sämtliche Transaktionen speichert und ohne eine zentrale Partei verwaltet wird. Durch den Zugriff auf alle Transaktionen kann jeder Nutzer überprüfen, ob eine Geldeinheit bereits ausgegeben wurde. Das Schreiben in die Blockchain führt jedoch zu einem Leistungsengpass, da die Netzwerkteilnehmer regelmäßig Konsens über den Zustand der Blockchain erreichen müssen. Vor kurzem wurden Payment-Channel-Netzwerke [PD16] vorgeschlagen. Payment-Channel-Netzwerke sind ein Ansatz, der auf einer Blockchain aufbaut, aber Peer-to-Peer-Zahlungen ermöglicht, die nicht in die Blockchain geschrieben werden und daher nicht dem genannten Leistungsengpass unterliegen. Lightning ist eine Implementierung eines Payment-Channel-Netzwerks für Bitcoin, die zunehmend genutzt wird. In der vorliegenden Dissertation untersuchen wir Sicherheitsaspekte von Payment-Channel-Netzwerken am Beispiel von Lightning.

Wir beschäftigen uns mit dem Lightning-Protokoll und entwickeln einen Ansatz zur maschinellen Überprüfung seiner Sicherheit. Wir beschäftigen uns mit der Schicht unter einem Payment-Channel-Netzwerk, zeigen Gestaltungsmöglichkeiten für diese Schicht, und analysieren empirisch das Peer-to-Peer-Netzwerk von Bitcoin. Wir beschäftigen uns mit der Nutzerseite von Payment Channels und entwickeln zwei Ansätze, mit denen Nutzer dritte Parteien einbeziehen können, um ihre Payment Channels zu sichern. Schließlich beschäftigen wir uns mit Anwendungen, die Payment-Channel-Netzwerke nutzen und entwickeln ein Protokoll für Ticketing im öffentlichen Personenverkehr, dessen Sicherheitseigenschaften auf einem Payment-Channel-Netzwerk basieren.

Die Sicherheit des Lightning-Protokolls wurde bereits untersucht. Im Gegensatz zu vorherigen Arbeiten streben wir eine maschinell überprüfte Überprüfung der Sicherheit von Lightning an und legen die Grundlagen für eine solche Überprüfung: Wir tragen eine formale Spezifikation der Sicherheitseigenschaft bei sowie eine formale Spezifikation des Lightning-Protokolls in der formalen Spezifikationssprache TLA$^+$. Wir entwickeln eine Kette von Abstraktionen von der Spezifikation des Lightning-Protokolls bis hin zur Sicherheitseigenschaft. Durch eine Kombination aus Model Checking und Beweisen, um die Sicherheit dieser Abstraktionen nachzuweisen, zeigen wir, dass das Lightning-Protokoll die Sicherheitseigenschaft erfüllt. Obwohl unsere Überprüfung auf endlichem Model Checking beruht, schafft sie eine belastbare Grundlage für einen zukünftigen vollständig maschinell überprüften Sicherheitsbeweis.

Ein Payment-Channel-Netzwerk setzt auf einer Basisschicht auf. Wir zeigen, dass diese Basisschicht nicht zwingend durch eine Blockchain realisiert werden muss, sondern auch durch ein Netzwerk von Banken oder eine digitale Zentralbankwährung implementiert werden kann. Das Lightning-Netzwerk beruht auf Bitcoin als Basisschicht. Vorherige Arbeiten haben, basierend auf Annahmen über das Peer-to-Peer-Netzwerk von Bitcoin, gezeigt, dass Bitcoin die Anforderungen erfüllt, die das Lightning-Netzwerk an seine Basisschicht hat. Es ist empirisch feststellbar, dass diese Annahmen vom aktuellen Peer-to-Peer-Netzwerk von Bitcoin erfüllt werden, allerdings ist wenig über das Peer-to-Peer-Netzwerk selbst bekannt. In einem empirischen Teil dieser Dissertation, sammeln und analysieren wir Indizien für bisher unbekannte Metriken über das Peer-to-Peer-Netzwerk, wie zum Beispiel die Zahl der unerreichbaren Peers im Netzwerk oder den Knotengrad von erreichbaren Peers. Wir stellen fest, dass im April 2026 ungefähr 50 000 unerreichbare Peers im Bitcoin Peer-to-Peer-Netzwerk sind. Wir stellen außerdem fest, dass über 50 % der erreichbaren Peers einen Knotengrad haben, der nah am Standardwert für das Verbindungslimit liegt, was nahelegt, dass diese Peers keine weiteren eingehenden Verbindungen akzeptieren können und die Verbindungskapazitäten des Netzwerks nahezu ausgeschöpft sind.

Mit den Vorteilen eines Payment-Channel-Netzwerks, wie die Möglichkeit, eine hohe Anzahl an Transaktionen zu verarbeiten, und einer geringen Latenz für Transaktionen, geht die Anforderung einher, die Basisschicht regelmäßig auf betrügerische Transaktionen hin zu überwachen und auf solche Transaktionen in einem gewissen Zeitfenster zu reagieren. Es wurden Ansätze vorgeschlagen, die es einem Nutzer erlauben, diese Aufgabe sicher an eine dritte Partei, einen sogenannten Watchtower, zu übertragen. Wir entwickeln zwei neuartige Watchtower-Ansätze. Der erste Ansatz macht Watchtower-Ansätze wirtschaftlich praktikabel, indem Trusted Execution Environments eingesetzt werden, um den Speicherbedarf auf Seiten des Watchtower-Anbieters zu verringern. Der zweite Ansatz beseitigt die Notwendigkeit einer direkten Interaktion zwischen Nutzer und Watchtower, wodurch die Abhängigkeit eines Nutzers von einem bestimmten Anbieter verringert wird, da ein Payment Channel grundsätzlich von jedem Watchtower geschützt werden kann.

Schließlich zeigen wir die praktische Relevanz von Payment-Channel-Netzwerken durch eine Anwendung im Kontext des öffentlichen Personenverkehrs. Im öffentlichen Personenverkehr ist es häufig so, dass mehrere Anbieter gemeinsam ein Verkehrsnetzwerk betreiben und Reisende einen Fahrschein bei einem Anbieter kaufen können und den Fahrschein für Strecken nutzen können, die von einem anderen Anbieter betrieben werden. Wir schlagen ein Protokoll vor, das ein Payment-Channel-Netzwerk für die Bezahlung von Fahrscheinen nutzt. Durch die Nutzung eines Payment-Channel-Netzwerks implementiert das Protokoll einen Fair Exchange eines Fahrscheins gegen eine Zahlung zwischen einem Kunden und einem Anbieter und jede Zahlung wird direkt an den Anbieter geschickt, dessen Fahrzeuge der Kunde nutzt.

Insgesamt zeigt die Dissertation die vielschichtigen Sicherheitsaspekte von Payment-Channel-Netzwerken auf und entwickelt Ansätze, um deren Sicherheit und praktische Nutzbarkeit zu verbessern. Damit leistet sie einen Beitrag zur Entwicklung zukünftiger Zahlungssysteme, die zugleich praktikabel, dezentral und skalierbar sind.

Contents

List of Figures

List of Tables

1. Introduction

Digital payments have become an integral part of everyday life. Recent studies reveal that nearly half of all payment transactions are conducted digitally [Ban24]. These payments are processed by central parties like banks, credit card companies, and payment service providers. One of these parties failing can have a severe impact rendering the processing of payments impossible [GJ24]. With the development of Bitcoin [Nak08], it was shown that a payment system could be built in a decentralized way without depending on central parties. In Bitcoin, all processed transactions are stored in a blockchain, a distributed data structure of which a copy is stored by the peers in an open peer-to-peer network. Storing all transactions in the blockchain limits the rate of transactions that can be processed so that Bitcoin can process only up to seven transactions per second [Cro+16]. This is in stark contrast to actively used systems such as Visa which processed in 2025 on average more than 8000 transactions per second [Vis25].

One proposed approach to increase the number of transactions that can be processed in a decentralized system is to build a second layer on top of a blockchain. The basic idea of such a second layer is that funds can be moved from the blockchain to the second layer, transactions can be processed inside the second layer, and funds can be moved back to the blockchain. The transactions that are processed inside the second layer are not stored on the blockchain. Thus, more transactions can be processed and the performance limits of the blockchain can be circumvented. The prevalent approach for such a second layer for payment-focused blockchains, such as Bitcoin, is to use payment channel networks. For blockchains that offer richer smart contract functionality, such as Ethereum [But14], rollups [TSH22] have emerged as the dominant scaling approach while other blockchains focus on approaches for scaling on the first layer, for example, via sharding [Wan+19a]. In this thesis, we focus on payment channel networks because they constitute a scaling approach that is particularly well suited to payments. Specifically, payment channel networks enable low-latency payment execution, provide immediate finality once a payment has been processed, and are economically viable even for payments of small value.

A *payment channel* is opened between two parties by sending funds on the blockchain to an account from which the two parties can only spend with the consent of both parties. Each party maintains a local record of how the funds locked in the channel are allocated between both parties. Whenever a payment is made within the channel, the parties mutually agree on an updated allocation of these funds. The channel can be closed by the two parties by publishing a jointly authorized transaction to the blockchain that distributes the locked funds according to the most recent allocation agreed upon by both parties. A *payment channel network* is formed by combining multiple payment channels between different

users. Such a network enables payments between users who do not share a direct channel by routing transactions along a path of intermediate channels. The Lightning Network [PD16] is a payment channel network that has been developed for Bitcoin. Lightning, the protocol behind the Lightning Network, has been implemented and the Lightning Network is actively used [Bar24]. Since users entrust their funds to the protocol, a highly relevant question is whether Lightning is secure – specifically, whether Lightning securely manages the users' funds and ensures that users can unilaterally and fully withdraw their funds at any time. However, being secure is not sufficient for Lightning to be useful; it must also be practical. By practicality, we refer both to the existence of meaningful real-world use cases and to the question whether the requirements it imposes on users are feasible and manageable in practice. Therefore, this thesis focuses on the Lightning Network as an example of a payment channel network with the motivating question:

Is Lightning secure and practical?

Whether Lightning is secure depends on Lightning itself and on the layer underlying Lightning. Therefore, we present in the following three specific research questions with respect to the security of Lightning, the layer underlying Lightning, and the practical applicability of Lightning.

1.1. Research Questions and Contributions

In this thesis, we investigate payment channel networks across three layers. We begin with a theoretical analysis of payment channel networks. We formally model Lightning and define a security property. Using model checking, we verify that the protocol satisfies this property.

Next, we consider the layer underlying Lightning. We explicitly define the assumptions that Lightning makes about its underlying layer. We demonstrate that these assumptions are not inherently tied to blockchain-based systems and could also be realized by systems implemented by banks or payment service providers. Further, we discuss to which extent these assumptions are satisfied by Bitcoin. The properties of Bitcoin depend on the its peer-to-peer network, however, the actual topology of the peer-to-peer network is largely unknown. We reduce this knowledge gap by contributing an empirical measurement of previously unknown metrics of the Bitcoin peer-to-peer network.

Finally, we turn to the layer above Lightning. We design protocols that enable users to meet the requirements that Lightning imposes on users and present a concrete application by developing a protocol that leverages Lightning to improve ticketing in public transport systems.

In the following, we detail these contributions and formulate the corresponding research questions.

1.1.1. Verification of Security Properties of a Payment Channel Network

The approach of payment channels requires users to restrict their freedom of controlling their funds by locking the funds inside a payment channel and handing over control of their funds to the payment channel protocol. To hand over the control of their funds to a payment channel protocol, users need to trust that the payment channel protocol is secure. Secure means above all that the protocol guarantees that the protocol correctly manages a user's funds and that each honest user can unilaterally withdraw and retrieve their funds. Because of the complexity of Lightning, it is difficult to assess whether Lightning is secure. Therefore, we pose our first research question:

How to verify that a payment channel network fulfills its security properties?

There is a range of methods to verify properties of a protocol from informal analyses to formal proofs. These methods differ in the effort required to apply them and the certainty that they provide with respect to the correctness of their results. For this thesis, we choose Lightning as an example of a payment channel protocol. We contribute a formal specification of Lightning in TLA$^+$ and formally specify the security property that an honest user finally retrieves the user's correct balance. We further contribute a method to assess the security of Lightning using model checking. While model checking per se is a highly-automated method and has a low effort to apply, model checking comes with the problem of state space explosion: When model checking Lightning, the states of every possible execution of Lightning are calculated and compared against a security property. Because there is an infinite number of possible executions, it is not possible to enumerate all executions. Therefore, we model check Lightning only for models with a small number of users and payments that the users perform. However, because of the complexity of Lightning, it is challenging to use model checking even for small models. We contribute a stepwise refinement approach with which we reduce the problem of model checking the whole Lightning protocol to problems that are smaller and feasible to model check. With the stepwise refinement approach, we show by model checking that our formalization of Lightning fulfills our formalization of the security property for a set of specific models, including scenarios with two concurrent payments and multi-hop payments over up to six hops. This gives strong confidence that the formalization fulfills the security property and that it is worth pursuing the goal of a complete machine-checked proof in future work.

1.1.2. Validation of Assumptions of a Payment Channel Network

The approach that we use to verify the security of a payment channel network is based on an abstraction of the layer underlying the payment channel network. This abstraction specifies the assumptions that the payment channel network makes about the underlying layer. To apply the verification results in practice, it is relevant whether the layer underlying a payment channel network actually implements the abstraction used for the security verification. Therefore, we pose the research question:

> Are the assumptions that payment channel networks make about their underlying layer valid?

We approach this question again by using Lightning as an instance for a payment channel network. We explicitly define the assumptions that are made by Lightning with respect to the layer below with a particular focus on formulating these assumptions as minimally as possible. While Lightning is built upon Bitcoin, we show that Lightning requires only a subset of the properties offered by Bitcoin. We further find that the assumptions of Lightning about its underlying layer could be implemented also by other systems that could replace Bitcoin as a first layer below a payment channel network.

We continue by discussing whether Bitcoin actually fulfills the properties that are required by Lightning. Indeed, previous work has proven that Bitcoin meets properties that can be mapped to the properties that are required by Lightning. However, these proofs again depend on assumptions. One of these assumptions is that information propagates quickly in the peer-to-peer network of Bitcoin. Our measurements [NGH] of the time it takes for blocks to be propagated by the peer-to-peer network of Bitcoin show that typically the delay for block propagation is within the required range. The propagation delay of blocks depends on the forwarding behavior of the peers and the topology of the peer-to-peer network, specifically the degree of peers, i.e. the number of connections that each peer has [SZT22a]. While we do not know exactly the behavior of each peer, the large majority of peers runs the Bitcoin reference client[1] whose behavior can be inspected. However, not much is known about the topology of the peer-to-peer network. Thus, to understand the reasons for the propagation delay and to be able to assess the effects of changes in the behavior of a client on the propagation delay of blocks, we need more knowledge on the topology of the peer-to-peer network.

The peer-to-peer network of Bitcoin is an unstructured network but peers can be classified into reachable peers that accept incoming connections and unreachable peers to which other peers cannot open connections. Reachable peers play an important role because these are the only peers that can be connected to. We contribute the first measurement of the degree of a large set of reachable peers in the peer-to-peer network showing that many peers have a higher degree than expected by previous work: About 50 % of reachable peers have a peer degree of about 125 which is the default configuration for a peer's connection limit. We experimentally validate the interpretation that more than 50 % of the reachable peers are close to their connection limit and therefore restrict incoming connections.

Unreachable peers connect to reachable peers and, thus, the peer degree of reachable peers depends on the number of unreachable peers in the network. While the number of reachable peers can be measured, it is inherently difficult to count the number of unreachable peers because it is hard to differentiate an unreachable peer from a non-existing peer. We contribute new heuristics to estimate the number of unreachable peers, apply the heuristics, and evaluate them. In April 2026, our observations show about 48 000 unreachable peers in the Bitcoin peer-to-peer network. With the numbers of reachable and

[1] See `https://www.dsn.kastel.kit.edu/bitcoin/#useragents`

unreachable peers and our results of the peer degree, we contribute previously unknown metrics of the Bitcoin peer-to-peer network. These metrics can be used to parameterize models of the peer-to-peer network more realistically. However, to simulate or analytically calculate the propagation delay in the network, more metrics are needed, e.g., the peer degree of unreachable peers and latencies between the peers. Gathering or estimating these metrics is a challenge for future research.

1.1.3. Building Extensions and Applications for a Payment Channel Network

Having shown how to verify the security of a payment channel network and having examined the layer underlying a payment channel network, we turn to the application of payment channel networks in practice. Being secure is a necessary but not a sufficient condition for a protocol to be applied in practice. In practice, it is also relevant that the requirements that a protocol puts on users running the protocol are implementable for users and that the protocol provides plausible benefits. Therefore, we pose the research question:

> How can payment channel networks be applied in practice?

A requirement for users running a payment channel protocol is that users need to regularly watch the blockchain and react to cheating attempts that are possibly made by the other party in the channel. This requirement can be a hindrance to use payment channels if users do not have a reliable network connection or expect to have a longer downtime. It is possible to transfer the fulfillment of this requirement to a third party, called a watchtower. We contribute two approaches for watchtowers that come with different trade-offs. In the first approach, we use trusted execution environments to restrict the actions of the watchtower provider and, in particular, allow data to be stored by the watchtower provider that cannot directly be read by the provider. This approach allows for efficiently storing secrets at the watchtower provider that are required for watching and reacting to outdated closing attempts. In the second approach, we use IPFS (InterPlanetary File System) [Ben14], a decentralized file system, to store the data required for watching at a third party and making it publicly accessible so that even watchtower providers who were not explicitly engaged can assume the role of the watchtower for a payment channel.

Finally, we explore how payment channel networks can be used in a protocol for ticketing in public transport and how the benefits of payment channel networks can be exploited. Public transport systems are often systems with multiple transport providers. In such systems, passengers may purchase a ticket from one transport provider while using vehicles operated by another provider. As a result, the provider issuing the ticket may collect revenue on behalf of other providers, necessitating subsequent revenue clearing among the involved providers. While previous work has explored how to build ticketing systems for public transport [Gud15; Hin15; Han+21] and how to clear revenue between providers [Rin86; TA06; Hua+10; Yic20], the novelty of our approach lies in addressing both challenges within a single unified protocol. Using a payment channel network for

ticket payment enables sending each payment directly to the provider earning the revenue which obviates the need for a later clearing. Further, payment channel networks improve the security of ticket payment by enabling a fair exchange of a ticket against the payment for the ticket, i.e., the buyer of a ticket receives a valid ticket if and only if the ticket is being paid for. This mechanism provides a benefit for public transport providers as it protects them against fare evasion and it enables travelers to prove that they have paid for a ticket. Although payment channel networks impose certain limitations such as the need to regularly rebalance channels that are predominantly used for unidirectional payments, the ticketing use case demonstrates that payment channel networks constitute a valuable building block for achieving higher-level security properties in application protocols.

1.2. Thesis Outline

The remainder of this thesis is structured as follows.

In Chapter 2, we introduce Bitcoin and explain payment channel networks and, in particular, present Lightning, the protocol of the Lightning Network. We also give an overview of previous research related to payment channel networks.

In Chapter 3, we introduce TLA$^+$and present our formalization of a security property for Lightning and show how we formalize Lightning in TLA$^+$.

In Chapter 4, we present our stepwise refinement approach to facilitate model checking the security property of Lightning and show how we use model checking to check that the formalization of Lightning fulfills the security property.

In Chapter 5, we contribute a model for a generalized first layer for a payment channel network and present our empirical results that extend our knowledge of the topology of the Bitcoin peer-to-peer network.

In Chapter 6, we present two approaches for watchtowers and a protocol for ticketing in public transport that uses a payment channel network for payment to achieve a fair exchange of tickets and payments.

We conclude this thesis and give an outlook in Chapter 7.

1.3. Underlying Publications

This thesis presents and extends work that has been previously published in the publications listed below.

Chapters 3 and 4:

- M. Grundmann and H. Hartenstein. Verifying Payment Channels with TLA+. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1-3, May 2022.
- M. Grundmann and H. Hartenstein. Security of the Lightning Network: Model Checking a Stepwise Refinement with TLA+. In *Integrated Formal Methods (iFM)*, pages 313–335, Nov. 2025.

Chapter 5:

- M. Grundmann and H. Hartenstein. Fundamental Properties of the Layer Below a Payment Channel Network. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology (CBT)*, pages 409–420, Sep. 2020.
- M. Grundmann, H. Amberg, M. Baumstark, and H. Hartenstein. Short Paper: What Peer Announcements Tell Us About the Size of the Bitcoin P2P Network. In *Financial Cryptography and Data Security (FC)*, pages 694–704, May 2022.
- M. Grundmann, M. Baumstark, and H. Hartenstein. On the Peer Degree Distribution of the Bitcoin P2P Network. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1-5, May 2022.

Chapter 6:

- M. Leinweber, M. Grundmann, L. Schönborn, and H. Hartenstein. TEE-Based Distributed Watchtowers for Fraud Protection in the Lightning Network. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology (CBT)*, pages 177–194, Sep. 2019.
- H. Bönisch and M. Grundmann. Decentralizing Watchtowers for Payment Channels using IPFS. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 27–32, Jul. 2022.
- M. Grundmann, O. von Zastrow-Marcks, and H. Hartenstein. On the Applicability of Payment Channel Networks for Allocation of Transport Ticket Revenues. In *2022 International Conference on Computer Communications and Networks (ICCCN)*, pages 1-7, Jul. 2022.

2. Fundamentals and Related Work

The concept of payment channel networks was proposed with the Lightning Network, an implementation of a payment channel network on top of Bitcoin. In the first part of this chapter, we introduce the Lightning Network and explain the main ideas behind its protocol called Lightning. We start by explaining basic terms that are used throughout this thesis and, before introducing Lightning, we present fundamentals about Bitcoin that Lightning is based on.

In the second part of the chapter, we give an overview of related work in the context of payment channel networks and discuss how our contributions fit into the related work. To give an impression of the wide scope of research in the field of payment channel networks, we introduce alternative approaches to implement payment channel networks, improvements and extensions to payment channel protocols, and empirical research on the usage of payment channel networks.

2.1. Basic Terms

Blockchain While initially introduced as a term to describe the data structure used by Bitcoin, the term 'blockchain' has been used in the literature with a broader scope [TVV23], e.g. referring to variations and generalizations of the initial data structure or to systems maintaining such a data structure. We use blockchain as a term to refer to the data structure. A blockchain is based on blocks. A block is a data structure that consists of a payload and a header. The payload of a block can, for example, consist of a set of transactions. The block header contains metadata of the block and is used to link the block to other blocks. A blockchain is a data structure consisting of a set of blocks in which all but one blocks reference exactly one other block. The block that a block references is called parent block. The reference is implemented by including in the block the hash value of the parent block. The only block that does not reference another block is the first block in the chain, called genesis block. Each block can contain additional data. The *height* of a blockchain is the number of blocks in the blockchain. The height of a specific block b is the number of blocks in the prefix of the blockchain from the genesis block to the block b.

Payment Channel A payment channel is a contract between two parties that defines how a certain amount of funds is jointly managed. When the contract is created, both parties can deposit funds into the contract. It is recorded in the contract, which amount of

the contract's funds belong to each party. The contract guarantees that each party can close the contract. When the contract is closed, each honest party is guaranteed to receive at least the party's share of the contract's funds. The contract allows for the participating parties to agree on a new allocation of how funds are distributed between the parties. This property to agree on a new allocation of funds is the property that makes a payment channel useful: A payment channel can be used to send a payment from one party to the other party by both parties agreeing on a new allocation of funds that represents the updated balances after the payment.

A payment channel is an instance of an off-chain payment system. The distinctive properties of a payment channel are that a payment channel is a protocol between two parties, the state changes are limited to payments, and that the blockchain communication is limited to opening and closing the channel. There are also approaches generalizing from this strict definition building multi-party payment channels with more than two parties (see Section 2.4.2.3), state channels that allow for more general state changes than just payments (see Section 2.4.2.2), or payment channel approaches that use additional blockchain communication while a channel is open (see Section 2.4.2.4).

Payment Channel Networks A payment channel network is a graph consisting of users as vertices and payment channels as edges. Users can send multi-hop payments over a path in a payment channel network and thereby send payments to users they do not share a payment channel with. Intermediary hops forward payments by receiving an incoming payment in one channel and sending an outgoing payment in another channel. The protocol for multi-hop payments must ensure that a multi-hop payment is atomic which means that the receiver receives the payment if and only if the sender sends the payment and no intermediary hop can steal or lose funds during the payment.

2.2. Bitcoin

While digital cash was proposed in the 1980s [Cha83; CFN88], the early approaches relied on central parties like banks to validate that a received digital coin was actually valid and has not already been spent. The innovation of Bitcoin [Nak08] was to store all payments in a decentralized data structure called a blockchain that is protected by a mechanism called proof of work. As all transactions are recorded in the blockchain and because the blockchain is public, everyone can check the validity of transactions and which units of money have already been spent.

Bitcoin can be divided into three logical levels: The lowest level is a peer-to-peer (P2P) network that is open for everyone to participate. This network is used to distribute information between the peers and maintain the blockchain. On a middle level, the consensus level, the peers (called miners) build the blockchain and achieve probabilistic

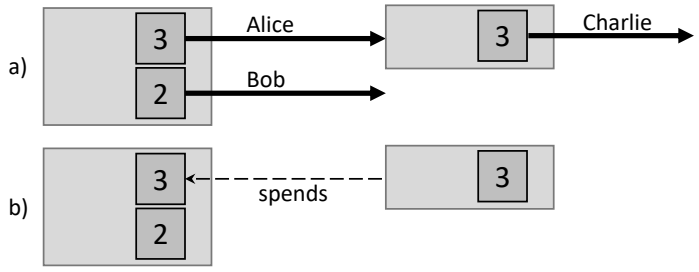

Figure 2.1.: Two ways of drawing two transactions in Bitcoin. The transaction on the left has two outputs of which the upper output is spendable by Alice and the lower output is spendable by Bob. The transaction on the right spends Alice's output of the transaction on the left and has an output that is spendable by Charlie. To simplify, we do not explicitly draw transaction inputs. The amounts of each output are noted inside each output. There are two versions showing the same two transactions. Version a) illustrates who can spend each output by drawing outgoing arrows at each output labeled by the users who can spend the output. Version b) illustrates with the dashed arrow that the input of the transaction on the right references and spends the first output of the transaction on the left.

consensus[1] over a prefix of the blockchain. The contents of the blockchain constitute the application level. On this level, users can create transactions. A transaction is a data structure that specifies the transfer of funds (see next paragraph). Users can publish transactions which are considered to be confirmed as soon as they are included in the prefix of the blockchain over which the peers have consensus. A payment channel is created using transactions on the application level. Therefore, this section focuses on the application level. In Chapter 5, we introduce more details on the network level of Bitcoin and present our results of measurements on the network level.

A payment system must store the allocation of funds to users. In Bitcoin, the allocation of funds to users is stored by storing the history of each unit of the currency. The unit to represent funds in Bitcoin are bitcoins. Once created, bitcoins can be spent using transactions. On an abstract level, a transaction consumes a set of bitcoins owned by the sender of a transaction and produces new bitcoins that are owned by the receiver of a transaction (see Fig. 2.1). Technically, a transaction is a data record that contains a set of *inputs* and a set of *outputs*. Each output has an amount and a set of conditions that need to be met to spend the output. Each input refers to an output in a previous transaction that is spent by the input.

An output can have multiple *spending conditions*. To spend an output, an input has to fulfill one of the output's spending conditions. The spending conditions are implemented using a stack-based script language called Script. The language Script allows for implementing simple conditions such as the requirement for the spending transaction to be signed by a specific key or by a key with a specific hash value. Other possible conditions are conditions that require signatures from multiple keys or the preimage to a specific hash. Further, outputs can have an absolute timelock that enforces that an output can be spent only

[1] Consensus is achieved probabilistically because it is possible that two peers have different prefixes of the blockchain that are both valid. The probability for such an outcome falls exponentially with the number of blocks following the prefix of the blockchain over which consensus is achieved.

after the absolute timelock has passed. The absolute timelock is specified either using a timestamp or using a blockheight h that determines that a spending transaction can only be in a block that has at least height h in the blockchain. Outputs can also have a relative timelock. A relative timelock δ in an output o specifies that a transaction spending the output o can only be included in a block that has a height that is at least δ blocks higher than the height of the block that includes the transaction that contains the output o.

This way to model transactions is referred to as the UTXO model where UTXO stands for unspent transaction output. An unspent transaction output is an output that is not referenced by any input. In the UTXO model, the balance of a user is the sum of the amounts of UTXOs that are spendable by the user. Thus, a wallet of a user contains the secrets required to spend these UTXOs.

2.3. Lightning Network

A main challenge in implementing a payment system is to prevent double spending. This means to ensure that, once a sender has sent a payment of a unit of money to a receiver, only the receiver can spend the unit of money but the sender cannot directly spend the same unit of money again. Guerraoui et al. [Gue+19] show that preventing double spending of a particular unit of money requires consensus among all participants of the payment system who can confirm a transaction that spends the particular unit of money.

Reaching consensus becomes costly when there is a large number of participants that need to reach consensus because all of them can confirm such a transaction. Further, reaching consensus is costly when these participants can be malicious. Both factors apply to Bitcoin because, in principle, every participant in Bitcoin could be a miner and confirm any valid transaction and because it is assumed that participants can be malicious. One way to reduce the cost of reaching consensus is to restrict the set of participants in the payment system that can confirm transactions that spend a particular unit of money.

This principle is the main idea behind second layer protocols that are built on top of a blockchain. In such protocols, funds are first locked on the blockchain after which only a smaller set of participants can confirm transactions that spend these locked funds. One approach for such a second layer protocol is to use payment channels in which there are only two participants that manage funds that are locked on the blockchain. Because there are only two participants, reaching consensus is fast and, if the two participants fail to reach consensus, the blockchain serves as a fallback mechanism for resolving disputes. In 2015, Poon and Dryja [PD16] proposed a protocol for the Lightning Network which is a network of payment channels. This protocol, called Lightning, has since undergone substantial development and is now specified in a public Github repository [Lig26c].

In this section, we give an overview of how payment channels are implemented by Lightning and how multi-hop payments are performed. Lightning also defines the protocol for a peer-to-peer network describing how peers communicate with each other and how information is distributed in the peer-to-peer network. For our presentation in this section,

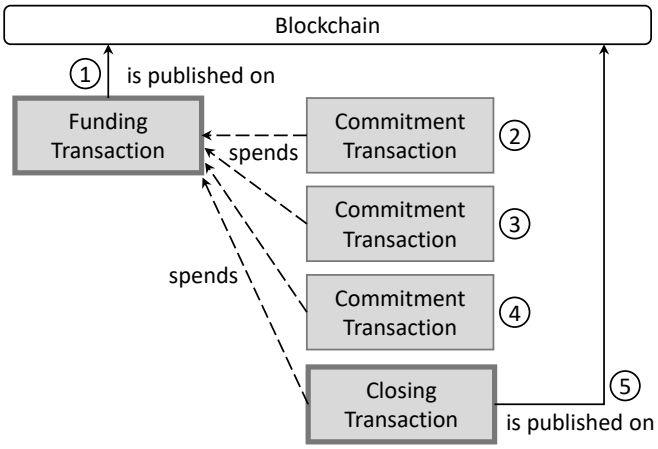

Figure 2.2.: Overview of transactions and their relations in Lightning. The funding transaction is published on the blockchain ① which is indicated by a bold border. The output of the funding transaction is referenced by the inputs of commitment transactions and the closing transaction. Because an output can only be spent once, only one of these transactions can be published on the blockchain. The figure shows the usual way that the channel is closed by publishing a closing transaction ⑤. While the figure shows only three commitment transactions, there can, in theory, be an unbounded number of commitment transactions.

we simplify Lightning and we focus on the aspects of Lightning that build the core of the protocol and are relevant for the remainder of this thesis. Further details of Lightning are discussed in Chapter 3 in which we present our formal model of Lightning. The peer-to-peer network aspects and other aspects such as how peers exchange metadata about payments and find routes for payments are left out of scope.

2.3.1. Overview

We first give an overview of the key ideas and mechanisms in Lightning and then present the protocol in more detail. We refer to the two users that participate in a payment channel as the channel's participants. A channel is opened by publishing a *funding transaction* on the blockchain (see Fig. 2.2, ①). A funding transaction contains an output, referred to as the *funding output*. We introduced a payment channel above as a contract that defines how the two participants manage a certain amount of funds. These managed funds are represented by the funding output. To record how much of the funds belong to each participant, *commitment transactions* are created (see Fig. 2.2, ②). A commitment transaction spends the funding output. If there are no ongoing payments, a commitment transaction contains an output for each of the two participants. The amount of each output corresponds to the balance of the respective participant. When payments are made over the channel, new commitment transactions are created that represent the updated balances (see Fig. 2.2, ③ and ④). When the payment channel is closed, the commitment transaction or a specific closing transaction is published on the blockchain (see Fig. 2.2, ⑤). The closing transaction distributes the funds deposited in the channel to the channel's participants.

The funding output is spendable only by transactions that are signed by both participants. To achieve that each participant can close the channel independently of the other participant, each participant has the other participant's signature for the commitment transaction. We say that each participant *holds* the commitment transaction to say that each participant has a version of the commitment transaction with the signature of the other participant so that the participant could sign and publish the transaction. Thus, each participant can close the channel by signing the commitment transaction and publishing the commitment transaction with both participant's signatures.

When one of the participants sends a payment to the other participant, both participants agree on a new allocation of the channel's funds and create a new commitment transaction representing the new allocation of funds (we describe the process later in more detail). While an honest participant may only publish the latest commitment transaction, a dishonest participant could publish an outdated commitment transaction that assigns the dishonest participant more funds than the latest commitment transaction. To disincentivize participants from publishing outdated commitment transactions, Lightning uses the following mechanisms: First, both participants hold different versions of the commitment transaction which makes it possible to distinguish who published a commitment transaction. Second, the outputs of an outdated commitment transaction can be spent by the other participant. Each output of the commitment transaction has an additional spending condition that allows the honest participant to spend the output if the commitment transaction is outdated. This mechanism disincentivizes dishonest behavior because a dishonest participant risks losing even the dishonest participant's funds to the other participant.

Figure 2.3 shows exemplary commitment transactions of one participant for different states and Fig. 2.4 shows the two different versions of both participants for one state. To implement that an outdated commitment transaction can be revoked but a current commitment transaction cannot be revoked, Lightning uses revocation keys that are shared when a transaction becomes outdated: Each output in the commitment transaction of one participant is spendable by the other participant using a revocation key for this specific commitment transaction (r_n^{A} in Fig. 2.3). Each user generates the revocation keys for the user's own commitment transactions. If a commitment transaction becomes outdated because a new commitment transaction is created, the revocation key is shared with the other party. If a dishonest user publishes an outdated commitment transaction on the blockchain, the commitment transaction's output for the dishonest user's balance is spendable by the dishonest user and by the other user using the revocation key. If both ways to spend the output were directly possible, there would be a race to spend the output between the dishonest user and the other user. To give the other user time to publish a transaction spending the outdated commitment transaction's outputs, the spending conditions of outputs of a user's own version of the commitment transaction that are spendable by the user themselves are timelocked using a relative timelock (Δt in Fig. 2.3). This timelock has the effect that the output can be spent by the dishonest user only a certain time after the commitment transaction has been included in the blockchain and, therefore, the other user has the opportunity to punish the dishonest user.

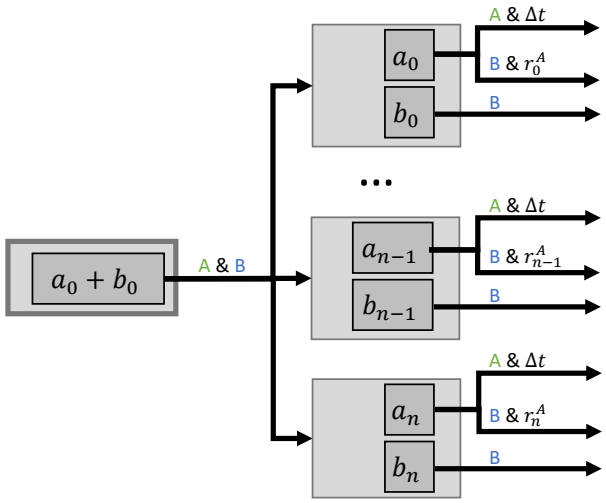

Figure 2.3.: Commitment transactions for different states of one user in Lightning. The transaction on the left is the funding transaction with a single output that is spendable by transactions that are signed by both users (A and B). On the right, multiple commitment transactions are shown. Each commitment transaction spends the output of the funding transaction and has two outputs, one for each user of the channel. The commitment transactions differ by the amounts of the outputs representing the allocation of funds in the channel at different states and each commitment transaction uses a different revocation key. The commitment transactions that are shown here are the versions that are held by user A. Thus, the output that is spendable by user A is timelocked (Δt) and contains an alternative spending condition that can be used by user B to revoke the transaction (B & r_n^A). The output for user B can be spent directly by user B.

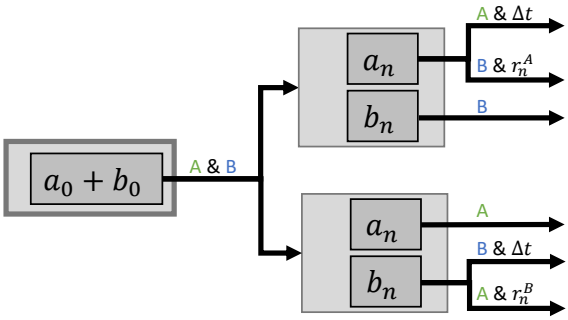

Figure 2.4.: Commitment transactions of both users in Lightning at the same state. Both commitment transactions reference the output of the funding transaction and, thus, only one of these transactions can be published on the blockchain. The upper commitment transaction is the version held by user A, the lower commitment transaction is the version held by user B. The two commitment transactions differ in that the output for user u in the version held by user u is timelocked (Δt) and can be revoked by the other user.

To summarize, the mechanisms used by Lightning and their purposes are listed in Table 2.1. The core idea of Lightning is to manage a tree of transactions of which some transactions can be published on the blockchain to enforce the state of the payment channel. In the following, we present the protocol to create and maintain this tree of transactions.

Table 2.1.: Overview of mechanisms used by Lightning for a payment channel and their purpose

Purpose	Mechanism
Each participant can close the channel independently of the other participant	Each participant has the other participant's signature for the commitment transaction
Disincentivize publication of outdated commitment transactions	Outputs in outdated commitment transactions can be spent by the other participant
Determine who published an old commitment transaction	Different version of commitment transaction for each user
Distinguish outdated and current commitment transactions	Participants send each other the keys to spend the outputs of outdated commitment transactions
Prevent race between cheating and honest participant	Outputs for cheating participant are timelocked with a relative timelock, honest participant can spend directly

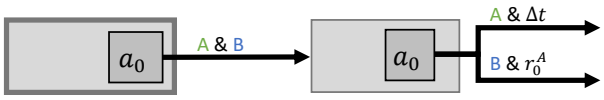

Figure 2.5.: Lightning's transaction tree directly after a channel has been opened. The funding transaction is published on the blockchain and there is an initial commitment transaction that sends the channel's capacity back to the funder (user A). The initial commitment transaction contains only a single output because the channel is single-funded and initially, the funder owns the entire capacity of the channel.

2.3.2. Opening a Payment Channel

If two parties agree to open a payment channel, they lock funds using a funding transaction. In the first versions of the protocol, a channel could only be single-funded, i.e. one of the users owns initially all funds of the channel. Recently, Lightning was extended to allow for both parties to put funds into the payment channel. Here, we describe only the simpler case of single-funded channels.

Opening a payment channel is initiated by a user who puts the funds into the channel and is called the *funder*. The other user is commonly referred to as the *fundee*. The funder sends a message to the fundee with the funder's channel parameters. These include the capacity of the channel which is the amount of funds that the funder wants to deposit into the channel, cryptographic key material, and parameters for the later channel operation, e.g., a maximum of the number of payments that the payment channel may concurrently process. If the fundee agrees to open a payment channel with the received parameters, the fundee sends a message with the fundees cryptographic key material and channel parameters to the funder. The funder creates the funding transaction which is a transaction that contains inputs that spend UTXOs of the funder and one single output. The output of the funding transaction has a single spending condition that can be fulfilled by a transaction that is signed by both the funder and the fundee. The funder creates an initial

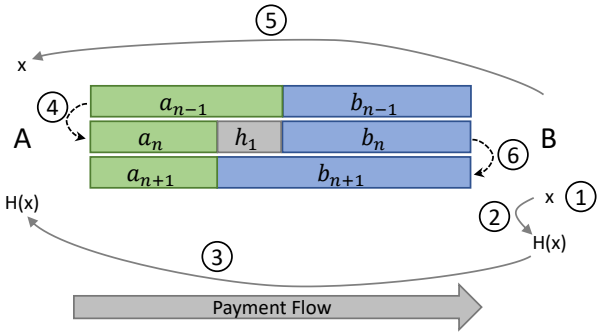

Figure 2.6.: Visualization of the steps of the protocol for payments over a payment channel. The figure shows a payment that is sent from user A to user B. The bars in the center of the figure visualize commitment transactions. Each segment of a bar represents an output where the width of each segment corresponds to the amount of the output. In the shown setting, the commitment transaction in state $n - 1$ has two outputs, one output for user A and one output for user B. In step ④, a new commitment transaction for state n is created with a third output h_1 for the HTLC. The amount of the HTLC is deducted from the balance of user A. In step ⑥, a new commitment transaction for state $n + 1$ is created where the HTLC output is removed and the amount of the HTLC is added to the balance of user B. The figure shows a successful payment. If the payment had failed, the step ⑥ would lead to a commitment transaction for state $n + 1$ that equals the commitment transaction for state $n - 1$.

commitment transaction with an input that spends the funding transaction's output and a single output that is spendable for the funder (see Fig. 2.5). The funder signs the initial commitment transaction and sends the signature of the initial commitment transaction to the fundee together with the transaction id of the funding transaction. With the funding transaction's id, the fundee also builds the initial commitment transaction and verifies the signature received from the funder. The fundee also creates a signature on the initial commitment transaction and sends the signature to the funder. Once the funder has received the signature on the initial commitment transaction, the funder publishes the funding transaction on the blockchain. When the funding transaction has been included in a block, the users of the payment channel exchange messages to notify each other that the channel is ready to operate.

Once the channel is open, both users have a signature of the other user on the initial commitment transaction. Because the initial commitment transaction spends the funding output and the funding output can be spent by a transaction that is signed by both users, each of the users could close the channel by signing and publishing the initial commitment transaction.

2.3.3. Payments over a Payment Channel

To make a payment over a payment channel, a primitive called Hashed Timelock Contract (HTLC) is used. An HTLC is a contract between the sender and the recipient of a payment that is defined as follows:

Definition 1 (Hashed Timelock Contract)**.** A *Hashed Timelock Contract (HTLC)* between two users, a sender and a recipient, is parameterized by an *amount*, a *hash value*, and a *timelock*. The contracts states that the recipient receives the specified *amount* of coins from the sender if the recipient provides a preimage to the specified *hash value* before the specified *timelock*. An HTLC can be *resolved* in two ways: If the recipient provides the preimage before the timelock, the HTLC is *fulfilled*. If the timelock passes before the HTLC is fulfilled, the HTLC has *timed out*. For the sender, the HTLC is called an *outgoing* HTLC; for the recipient, the HTLC is called an *incoming* HTLC.

A payment over a channel follows the following procedure (see Fig. 2.6). For each payment, the channel is updated twice: once for adding an HTLC and once for removing the HTLC again. We explain below how such an update is performed.

1. The recipient draws a random value x (see Fig. 2.6, 1).
2. The recipient calculates the hash value $H(x)$ and sends $H(x)$ to the sender (see Fig. 2.6, 2 and 3).
3. The sender updates the payment channel to include an HTLC for the payment's amount, the hash value $H(x)$, and a reasonable timelock that leaves sufficient time for the following steps. With the update of the channel, the HTLC's amount is deducted from the sender's balance (see Fig. 2.6, 4).
4. The recipient sends the value x to the sender thereby fulfilling the HTLC (see Fig. 2.6, 5).
5. The channel is updated to a state with the HTLC removed and the HTLC's amount added to the recipient's balance (see Fig. 2.6, 6).

2.3.3.1. Channel Update

A channel is updated to add or remove an HTLC in the following way. The steps are visualized in Fig. 2.7. To update the channel when adding an HTLC, the sender of the payment creates a new commitment transaction. In contrast to the previous commitment transaction, the new commitment transaction contains an additional output for the HTLC and the balance of the payment's sender is reduced by the HTLC's amount. The output for the HTLC has the HTLC's amount and a spending condition modeling the HTLC, i.e., the output is spendable for the recipient of the payment if the preimage is provided and the output is spendable for the sender of the payment if the timelock has passed. The payment's sender signs the new commitment transaction and sends the signature to the payment's recipient (see Fig. 2.7, ②). At this point in the protocol execution, the recipient has the sender's signatures for the previous and the new commitment transaction and could sign and publish either of them. The recipient revokes the previous commitment transaction by sending the revocation key for the previous commitment transaction to the sender (see Fig. 2.7, ③). Having sent the revocation key, the recipient would risk losing their funds if the recipient would publish the previous commitment transaction. At

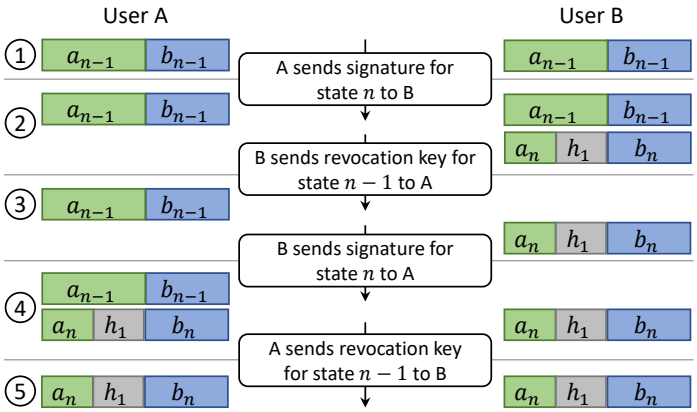

Figure 2.7.: Visualization of steps for the update of a payment channel in which an HTLC is added. The bars in the figure represent commitment transactions. Each segment of a bar represents an output of a commitment transaction. The column on the left shows the commitment transactions held by user A; the column on the right shows the commitment transactions held by user B. The figure shows the intermediate states for adding an HTLC for a payment from user A to user B. Initially, both users hold a commitment transaction that represents the state $n-1$ in which the commitment transactions contain only two outputs for the two users. In intermediate states, users hold two commitment transactions and could publish either of them on the blockchain. After has user revoked a state and, thus, cannot publish a commitment transaction on the blockchain anymore without risking to be punished, the bar representing the commitment transaction is not drawn anymore. Finally, both users hold a commitment transaction that includes an output for the HTLC.

this point, the channel has been updated only for one user: The recipient's commitment transaction includes the HTLC. However, the sender's commitment transaction does not yet contain the HTLC. To complete the update of the channel, the recipient also adds the HTLC to the sender's version of the commitment transaction, signs the commitment transaction and sends the signature to the sender (see Fig. 2.7, ④). The sender replies by sending the revocation key for the sender's previous commitment transaction to the recipient (see Fig. 2.7, ⑤). At this point, the commitment transactions of both users include the HTLC and the update has been completed.

Once the recipient has sent the preimage to the sender to receive the payment by fulfilling the HTLC, the recipient removes the HTLC from the channel and adds the HTLC's amount to their own balance. The process to update the channel is analogous to the process for updating a channel when adding an HTLC: The recipient sends a signature for the new commitment transaction to the sender. The sender replies with the revocation key for the previous commitment transaction and sends a signature for the new commitment transaction of the recipient to the recipient. Having received the signature, the recipient replies with the revocation key for the previous commitment transaction.

2.3.3.2. HTLCs in Commitment Transactions

Because each state of a payment channel must be enforceable on the blockchain, HTLCs must be encoded in commitment transactions. Figure 2.8 shows an example of a commit-

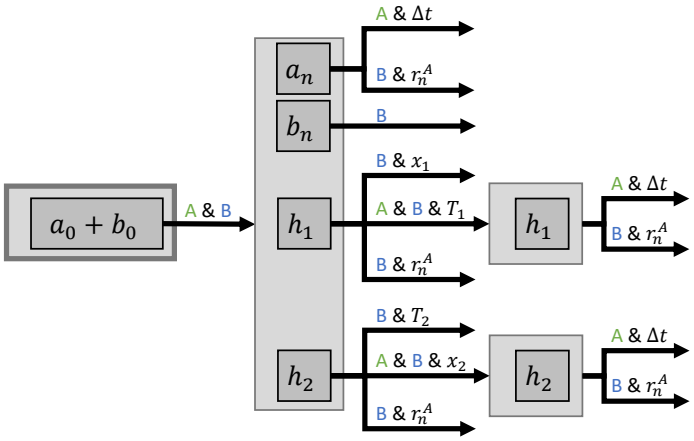

Figure 2.8.: Transaction tree showing a commitment transaction including one incoming and one outgoing HTLC. The figure shows the version of the commitment transaction that is held by user A. The output h_1 represents an HTLC from user A to user B. The output has three spending conditions: It can either be fulfilled by user B with the secret x_1, or by a transaction that is signed by both users after the absolute timelock T_1, or by user B and the revocation key associated with the commitment transaction. The HTLC timeout transaction for h_1 on the right spends the output using the middle condition and has itself an output that can be spent by user A after the relative timelock Δt or by user B with the revocation key. Analogously, the output for the incoming HTLC h_2 has three spending conditions: It can be spent by user B after the absolute timelock T_2 or by user A with the HTLC success transaction and the secret x_2.

ment transaction that contains two HTLCs – one incoming and one outgoing HTLC. For each HTLC, there is an additional output that encodes the conditions of the HTLC. These are, in principle, two conditions: One condition that allows the HTLC's recipient to spend the output if the recipient provides a preimage to the given hash value. And another condition that allows the sender of the HTLC to spend the output after the timelock. In practice, there are more spending conditions because, if a commitment transaction that includes an HTLC becomes outdated and is nevertheless being published on the blockchain, the HTLC output needs to be spendable by the honest party. To this end, each HTLC output has a third way of being spent which is the option to be spent by the counterparty if the counterparty knows the revocation key. However, simply having these three conditions to spend an HTLC output does not suffice: If an outdated commitment transaction includes an incoming HTLC for one of the participants, the participant could publish the commitment transaction and directly spend the HTLC output by providing the preimage. Similarly, if an outdated commitment transaction includes an outdated HTLC that has timed out, the participant could directly spend the HTLC output. To ensure that each HTLC can be fulfilled and timed out but that the HTLC output can also be revoked by the other party, Lightning uses additional transactions that are added to each commitment transaction for each included HTLC. For each incoming HTLC, an *HTLC success transaction* is created and, for each outgoing HTLC, an *HTLC timeout transaction* is created. These transactions are referred to as *HTLC second-stage transactions*. If a user sends a signature for a commitment transaction to the other user, signatures for all associated HTLC second-stage transactions are also sent. Each HTLC second-stage transaction has an output that has the same spending condition as the publishing party's output in the commitment transaction: The output

is spendable either by the publishing party after a relative timelock or by the counterparty with the revocation key. An HTLC second-stage transaction spends the respective HTLC output of the associated commitment transaction. To this end, the HTLC condition for the party who could publish the commitment transaction is spendable only by transactions that are signed by both parties of the channel. Further, an HTLC success transaction is only valid if the preimage of the hash value of the HTLC is given and an HTLC timeout transaction is only valid after the timelock of the HTLC. Figure 2.8 shows an exemplary transaction that contains an HTLC timeout transaction for the HTLC h_1 and an HTLC success transaction for the HTLC h_2.

2.3.4. Closing a Payment Channel

There are three ways in which a channel can be closed:

1. Unilaterally: A participant publishes their latest commitment transaction on the blockchain.
2. Bilaterally: Both participants create a specific closing transaction and publish it on the blockchain.
3. Dishonestly: A participant publishes an outdated commitment transaction on the blockchain.

The first way is the way for each party to close the channel independently of the other party. This approach has the disadvantage that the funds of the closing participants are locked for the length of the relative timelock. This time window is needed to give the other participant time to possibly revoke the transaction. If both participants cooperate, however, they can create a specific closing transaction whose outputs are immediately spendable (the second way). To this end, there is a specific protocol message that a participant can send to inform the other party that they want to close the channel. If this message is received, both parties wait for all current HTLCs to be removed from the channel and then exchange signatures for the closing transaction.

In the dishonest way (the third way), an outdated commitment transaction is published by one participant. The other participant has to regularly observe all published transactions to recognize this case. To facilitate the detection of outdated commitment transactions, each commitment transaction includes a *commitment number* that increases with each subsequent commitment transaction and each user stores the most recently assigned commitment number. Thus, if a transaction is observed that spends the output of the funding transaction, the commitment number that is included in the published transaction can be compared to the most recently assigned commitment number. If the commitment number in the observed transaction is lower than the stored commitment number, the observed transaction is an outdated transaction. In this case, the participant who observed the outdated transaction revokes the outdated commitment transaction. A participant *revokes* a transaction by creating a *revocation transaction* that spends the outputs of the transaction using the spending conditions that require a signature by the revocation key.

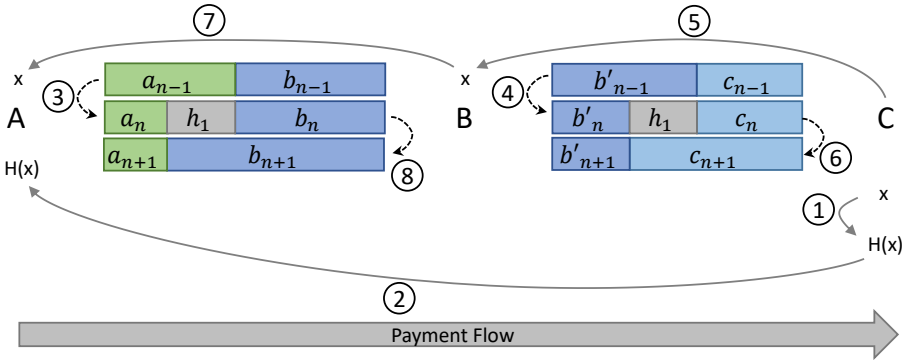

Figure 2.9.: Visualization of protocol steps during a multi-hop payment from user A over user B to user C. The bars between two users visualize the most current commitment transactions in the payment channel between the two users at different stages of the protocol. Each segment of a bar represents an output where the width of each output represents the amount of the output. For the payment from user A to user C, an HTLC is added to the channel between user A and user B and an HTLC is added to the channel between user B and user C. User B is part of two HTLCs: An incoming HTLC in the channel with user A and an outgoing HTLC in the channel with user C.

Thereby, the participant who published the outdated commitment transaction is punished and the punishing user receives the complete funds of the payment channel.

2.3.5. Multi-Hop Payments

Payment channels allow for payments between a channel's two participants. Using multi-hop payments, it is also possible for a user to send a payment to another user even if both users are not participants of the same payment channel. A *payment channel network* is a graph that has a vertex corresponding to each user an edge for each payment channel. Two vertices are connected by an edge if there is a payment channel between the two users that correspond to the two vertices. If there is a path between two users in the payment channel network, these two users can exchange multi-hop payments. The basic idea of a multi-hop payment over a path through the network is that the sender sends the payment's amount to the first intermediary, the first intermediary sends the payment's amount to the next intermediary, and the intermediaries forward the payment until the payment reaches the receiver of the payment. An inherent property of this approach is that a multi-hop payment can only be successful if each intermediary has a sufficient balance to send the payment's amount to the next hop.

Lightning ensures that a multi-hop payment is atomic, i.e. either all payments in the channels on the payment's path are successful or none of the payments. This ensures that the intermediaries in a multi-hop payment do neither lose funds nor steal funds. To ensure this atomicity, HTLCs are used as follows. The approach is a generalization of the payment approach for single-hop payments as described above. A multi-hop payment proceeds in the following steps. The steps are visualized in Fig. 2.9 at the example of a multi-hop payment with three users.

1. The recipient draws a random value x (see Fig. 2.9, ①).
2. The recipient calculates the hash value $H(x)$ and sends $H(x)$ to the sender (see Fig. 2.9, ②).
3. The sender updates their payment channel to the first intermediary to include an HTLC for the payment to the channel for the payment's amount and the hash value $H(x)$ and a timelock T_1 (see Fig. 2.9, ③).
4. Each intermediary i updates their payment channel to the next intermediary $i + 1$ to include an HTLC for the payment to the channel for the payment's amount and the hash value $H(x)$ and a timelock T_i (see Fig. 2.9, ④).
5. After the last intermediary has added the HTLC to the payment channel between the last intermediary and the recipient, the recipient sends the value x to the last intermediary thereby fulfilling the HTLC (see Fig. 2.9, ⑤). The payment channel is updated to remove the HTLC and the HTLC's amount is added to the recipient's balance (see Fig. 2.9, ⑥).
6. Each intermediary i sends the value x to the previous intermediary $i - 1$ thereby fulfilling the HTLC (see Fig. 2.9, ⑦). The payment channel is updated to remove the HTLC and the HTLC's amount is added to the intermediary $i - 1$'s balance (see Fig. 2.9, ⑧).
7. After the sender has received the value x and the HTLC in the payment channel between the sender and the first intermediary has been removed, the payment is completed.

For each payment, an intermediary has an incoming HTLC in one channel and an outgoing HTLC in another payment channel. These two HTLCs differ only by the values of their timelocks. The timelocks T_i are chosen in descending order from the sender to the recipient. Intuitively, the described approach ensures that intermediaries do not lose funds because once an intermediary's outgoing HTLC is fulfilled, the intermediary has $T_i - T_{i-1}$ time left to fulfill the corresponding incoming HTLC. We refer to this delta between the timelocks as the *forward delta*. The approach ensures that intermediaries cannot steal funds because intermediaries can only fulfill an incoming HTLC if they know the value x. Intermediaries can only learn the value x if their outgoing HTLC is fulfilled which means that they have paid the following intermediary.

2.3.6. Fair Exchange

An interesting property of HTLC-based payments in a payment channel network is that these payments provide an implementation for a specific variant of the fair exchange problem [Aso98; DM10]. In the fair exchange problem, there are two parties A and B who want to exchange items, i.e. user A wants to have user B's item and user B wants to have user A's item. A protocol implements a fair exchange, if the protocol guarantees the

properties effectiveness, fairness and timeliness [Aso98]. The properties can informally be described as follows:

Effectiveness If both users behave correctly and want to perform the exchange, then when the protocol has terminated, both users have exchanged their items.

Fairness If the protocol terminates and user A has received the item of user B, then user B has received the item of user A.

Timeliness If at least one user behaves correctly, this user can be sure that the protocol will have terminated at a certain point in time.

An HTLC-based payment as in Lightning is a protocol for a fair exchange of a payment against the preimage for a specific hash value. We briefly discuss why Lightning fulfills the three properties of a fair exchange (also shown by Ersoy et al. [Ers+23]): Lightning fulfills the effectiveness property because, if the sender and the receiver of a payment behave correctly and want to process the payment, the receiver sends the preimage to the sender and receives the payment. Lightning also guarantees the fairness property: The receiver of a payment receives the payment only if the receiver sends the preimage to the payment's sender and, if the receiver fulfills the HTLC before the payment's timelock, the receiver receives the payment. Lightning also fulfills the timeliness property. However, it might be that both parties have to wait for the payment's absolute timelock to pass until a payment has been resolved by being timed out. Nevertheless, the timeliness property is fulfilled because because both parties agree on the absolute timelock before starting the payment and the payment can be unilaterally completed after the absolute timelock.

If the parties involved in a payment are responsive, an HTLC-based payment is processed completely off-chain. However, if one of the parties becomes unresponsive or misbehaves, the blockchain is used as an arbiter. Such protocols that use a trusted third party in case of a dispute are called optimistic fair exchange protocols [DM10].

We conclude that HTLC-based payments provide an implementation of an optimistic fair exchange for the specific use case of an exchange of a payment in exchange for a preimage for a hash value. In Chapter 6, we will use this property to develop an application protocol for ticketing in public transport in which a customer and a transportation provider exchange a ticket in exchange for a payment.

2.4. Related Work

In this section, we give an overview of related work in the area of payment channel networks. The work can be divided into work that considers (1) approaches for payment channels other than Lightning, (2) generalizations of the idea of payment channels, (3) mechanisms for multi-hop payments, (4) management and security aspects, and (5) empirical results. We introduce each of these aspects within one of the following five subsections. For each aspect, we present exemplary research work and we do not strive for completeness. Instead, our goal is to give an impression of the broad landscape of the

research related to payment channel networks that our work is a part of. At the beginning of each subsection, we discuss the relation of the described work to our own work. For a more complete view on the field, we refer to surveys written with a focus on payment channel networks in particular [PT20; ZZS21; KSE21; KAV24], on second layer scaling solutions [Jou+19; Gud+20; Qi+25], and on general off-chain approaches [Wan+25].

2.4.1. Alternative Payment Channel Approaches

While Lightning has emerged as the main approach to implement payment channels on Bitcoin, there are also some alternative approaches. In the following, we present some of these approaches. By presenting the approaches, we aim to illustrate that the idea of payment channels does not necessarily mean that the channels are implemented by the Lightning protocol but that there are also other approaches. Thus, while we present in Chapters 3 and 4 a method for modeling a payment channel network and showing its security at the example of Lightning, there are other approaches for payment channel networks that could be analyzed in future work by transferring our approach. First, we introduce Duplex Micropayment Channels (DMC) because it is an approach that was an early alternative proposed to Lightning. One of the main reasons that Lightning was implemented instead of DMC is that a payment channel in DMC has a limited lifetime in contrast to a payment channel in Lightning that can stay open for an arbitrary amount of time. Then, we introduce Eltoo because it is an approach that can be considered a future successor of Lightning. Eltoo has a simpler approach to implement state updates but is currently not implementable because it relies on changes to the Bitcoin protocol. We go on by briefly presenting Teechain, an approach for payment channels that is fundamentally different because it relies on trusted execution environments (TEEs). In Chapter 6, we also use TEEs to make payment channels more secure but we use TEEs to improve watchtowers that monitor the blockchain and, if necessary, publish revocation transactions on behalf of payment channel users. Finally, we discuss payment channels on other blockchains such as Ethereum, Zcash, and Monero. This shows that the first layer of a payment channel network is not restricted to Bitcoin but can be exchanged for a different first layer. We further investigate this idea in Chapter 5 where we define the requirements of a first layer for a payment channel network and discuss alternatives to Bitcoin.

2.4.1.1. Duplex Micropayment Channels (DMC)

An early proposed alternative to Lightning are Duplex Micropayment Channels (DMC) [DW15]. While in Lightning old states are replaced by newer states by revoking the old states, DMC uses two different replacement mechanisms. A payment channel in DMC consists of a pair of two simple unidirectional payment channels. A unidirectional channel is funded by one party, the funder, and can only be used to send payments from the funder to the fundee. The unidirectional payment channels in DMC have a limited lifetime. The funder can only close the channel after the end of the channel's lifetime. The fundee can close the channel before the channel's lifetime has ended but the fundee is always

incentivized to close the channel in the newest state. Thus, this replacement mechanism is referred to as replace-by-incentive. A drawback of unidirectional payment channels is that they allow only the transfer of funds up to the capacity of the channel. Even if payments have been made on both of the two unidirectional channels, the channels need to be closed and reopened to be used again. A core idea of DMC is to allow for closing and opening the unidirectional channels off-chain. To achieve that, the unidirectional channels are funded by a funding transaction that itself is a transaction in the structure of an overarching payment channel. Once the unidirectional channels need to be closed and reopened, a new transaction is created in the overarching payment channel. For the overarching payment channel, DMC uses the replacement mechanism replace-by-timelock: The overarching payment channel has a limited lifetime and the channel can be closed only after the channel's lifetime has passed. Each transaction to initiate closing has a timelock that ensures that the transaction becomes valid only after the current lifetime of the channel. When a new state is created, the lifetime is decreased. When the lifetime of the channel has passed, the first transaction that becomes valid is the latest transaction and one of the users should publish this transaction on the blockchain to prevent the older transactions from being published when they become valid. This replacement mechanism has the disadvantage that the lifetime of a channel is limited and becomes shorter with each update of the channel. Because of the different replacement mechanisms used by DMC and Lightning, an advantage of Lightning is that channels have an unlimited lifetime while an advantage of DMC is that users do not need to access or watch the blockchain until the lifetime of the channel has passed. For a further comparison of DMC and Lightning, we refer to [McC+16].

2.4.1.2. Eltoo

The payment channel approach Eltoo uses a further replacement mechanism: replace-by-version. The idea behind this replacement mechanism is that each state is replaced by a newer state that carries a higher version number. This mechanism is used to build a payment channel approach as follows: An Eltoo payment channel is funded as in Lightning by a funding transaction with an output that is spendable only with signatures of both parties. For each state of the payment channel, there is an update transaction that can spend the funding output. An update transaction also has a state number that is increased with each new update. A significant difference between an update transaction in Eltoo and a commitment transaction in Lightning is that an update transaction has a single output. This single output is spendable in two ways: First, the update transaction's output can be spent after a relative timelock by a settlement transaction. For each update transaction, there exists a settlement transaction that spends the update transaction's output. The settlement transaction has two outputs that distribute the channel's balance to the participating parties according to the current balance distribution. Second, the output of an update transaction can also be spent immediately after publication by an update transaction that has a higher sequence number. This mechanism is the implementation of the replace-by-version idea: If an outdated update transaction is published on the blockchain, the other party can publish a newer update transaction that spends the outdated update transaction's output. After

the relative timelock has passed, the settlement transaction for the last published update transaction can be published and distributes the channel's funds to the parties.

The main limitation of the Eltoo proposal is that it is currently not implementable on Bitcoin. The replace-by-version mechanism requires that the input of the update transaction is not strictly bound to a specific transaction output. While the input must be signed by both parties to be published on the blockchain, the input must be able to spend either the funding transaction's output or the output of an older update transaction. Signing an input with such a 'floating' signature requires the introduction of the signature flag SIGHASH_ANYPREVOUT which signals that a transaction's inputs were not included in the signed data and, thus, the signed input can be attached to any previous output with a matching spending condition. While a proposal [DT17] for the addition of this signature flag exists, it is unclear whether or when this will be accepted by the Bitcoin community.

2.4.1.3. Teechain

A fundamentally different approach to deal with potential misbehavior is used by the Teechain [Lin+19] payment channel approach. Teechain introduces an intermediate layer between the blockchain and payment channels: In a network of Teechain nodes, each node runs a *treasury* inside a trusted execution environment (TEE). A TEE is a secure enclave that runs code whose integrity can be verified by remote attestation. Access to the TEEs data and code is protected by mechanisms built into the CPU hardware so that even the operating system cannot access the data or code inside the TEE. Teechain users deposit funds into a treasury and can use the deposited funds to open payment channels. The payment channels are managed by the treasuries: To make a payment, a user has to instruct their treasury to perform the payment and the treasuries of the channels involved in the payment run a protocol to perform the payment. To close a channel, a user instructs their treasury to close the channel. A user can withdraw own funds stored inside a treasury if the funds are not used as deposit for a payment channel.

2.4.1.4. Payment Channels on Ethereum

Payment channels were first proposed for the Bitcoin blockchain but they can also be built upon other blockchains. A blockchain with more powerful scripting capabilities as Bitcoin can facilitate the design of payment channels. Ethereum is an example of such a blockchain that can run more powerful smart contracts. The idea of payment channels has been transferred to Ethereum by the Raiden [Lab] project. The capabilities of Ethereum allow for using a replace-by-version approach. A payment channel is managed by an on-chain smart contract. By a call to the contract, a channel can be opened and funds are deposited. With each update of the channel, the channel parties exchange a signed state that encodes how the balances are distributed between the parties. The signed state also includes a state number that is incremented for every new state. To close a payment channel, one of the parties sends the latest signed state to the smart contract. The smart contract enters a challenge period in which the other party can send a newer state to the

smart contract with a higher state number. After the challenge period has ended, the smart contract closes the channel and pays the deposited funds to the two parties according to the latest state.

Sprites [Mil+19] is another payment channel approach based on the idea of Raiden but with an interesting additional property. While multi-hop payments in Raiden follow the same approach as in Lightning, Sprites makes use of the smart contract to reduce the maximum time that funds need to be locked during a multi-hop payment. In Lightning, each channel that forwards a payment must use a higher HTLC timelock than the next channel so that the hop between these two channels has enough time to forward a preimage after receiving the preimage. In Sprites, the same HTLC timelock can be used for all channels of a multi-hop payment because HTLCs are modified to refer to a central smart contract on the blockchain called the preimage manager. An HTLC is fulfilled on-chain if the preimage for the given hash is registered with the preimage manager. If one HTLC on a payment's route is fulfilled then all HTLCs on that payment route are fulfilled because the HTLCs in all channels refer to the same preimage manager.

2.4.1.5. Payment Channels on ZCash and Monero

In Bitcoin and Ethereum, all transactions are recorded on a public blockchain and the identity of a transaction's sender, the transaction's receiver, and the transaction's amount are publicly visible. The sender and receiver identities are pseudonyms which might be linkable to real world identities [BKP14]. To provide better privacy to users, other blockchains (e.g., Zcash, Monero) have been developed that hide the details of a transaction. To this end, Zcash uses zero knowledge proofs to prove that a transaction is correct without sharing a transaction's details. In Monero, to spend a transaction, the sender hides which output is spent by collecting other outputs from the blockchain and signing the transaction using a ring signature. A ring signature is a cryptographic signature that is created using a private key and can be validated using a set of public keys (called a 'ring') provided that the public key corresponding to the private key that was used for creating the ring signature is in the ring. It cannot be distinguished which of the public keys in the ring corresponds to the private key that has been used for signing. By using the ring signature, it becomes public that one of the outputs has been spent but it is not traceable which output has been spent. To ensure that each output is spent only once, each private key can be used only once to sign a transaction. Because the approaches of Zcash and Monero to implement transactions differ from the approach of Bitcoin, it is not trivial to transfer payment channel approaches that are designed for Bitcoin to Zcash and Monero. Still, some approaches have been proposed for implementing payment channels on Zcash [GM17] and Monero [Mor+20; Thy+22; Sui+22].

2.4.2. Generalization of Payment Channels

While we defined a payment channel above as a protocol with the properties that the protocol is for two parties, handles payments, and uses the blockchain only for opening

and closing a channel, approaches have been proposed that generalize a payment channel by giving up one or multiple of these properties. In the following, we present approaches that give up these properties for example for providing richer functionality and more efficient use of locked capital.

The following approaches that generalize the idea of payment channels represent different trade-offs compared to payment channel networks. Therefore, an approach based on a generalization of payment channels can be useful for applications in which payment channel networks do not fulfill an application's requirements. Such applications might, for example, be a payment second layer based on a first layer of a central bank digital currency (CBDC) as discussed in Chapter 5 or ticketing systems as discussed in Chapter 6.

Many results of research related to payment channel networks can be transferred to approaches that generalize the idea of payment channels. This also applies to the contributions of our work. Our contributions are not specific to payment channels but can, to varying degrees, be transferred to generalizations of payment channels.

2.4.2.1. Virtual Payment Channels

Payment channel networks are typically referred to as a second layer on top of a blockchain as the first layer. With virtual payment channels [Dzi+19b], an approach was introduced that creates a third layer on top of a payment channel network. The motivation of a virtual payment channels is to make recurring payments between two parties in a payment channel network more efficient. For recurring payments between parties who do not share a common payment channel, the intermediaries on the path through the payment channel network need to be involved in every payment. The idea of a virtual payment channel is that two users make multi-hop payment that is 'left open'. More specifically, a virtual payment channel is opened by locking coins on a path between the two users but not completing the payment. Now, the two parties can exchange payments by updating the state of the virtual payment channel which requires communication only between themselves. To close the virtual payment channel, the multi-hop payment is completed according to the final state of the virtual payment channel. While the first proposal [Dzi+19b] for virtual payment channels built on Ethereum, the idea has been transferred to Bitcoin by several follow-up works [JLT20; Aum+21b; KT21; Aum+23; Xie+23]. The optimization problem of where to open virtual payment channels was studied in [Aum+24].

2.4.2.2. State Channels

Payment channels were introduced for Bitcoin which has limited scripting capabilities. Other blockchains such as Ethereum, however, offer the possibility to execute feature-rich smart contracts. Building a payment channel on one of these blockchains with smart contract capabilities suggests to generalize payment channels to *state channels* [Col15; Mil+19] that allow for more general state changes than payments. A state channel is a protocol to replicate a state-machine between two parties without blockchain interaction

but using the blockchain to resolve disputes. Analogous to payment channels, state channels can also be used to build a network of state channels [CHX18; DFH18]. The idea of virtual channels can also be applied to state channel networks and state channels can be extended to allow for more than two parties to participate in a state channel [Dzi+19a].

2.4.2.3. Multi-Party Payment Channels

The restriction of payment channels to two parties can also be generalized. In a multi-party payment channel, multiple users collectively manage their funds and, without blockchain interaction, the users can process payments between each other and, possibly, with users in other multi-party channels. Recently, it has been shown that using multi-party channels for routing in payment channel networks can increase the payment success rate [Kot+25].

A trivial generalization of a two-party payment channel would be to involve all participating parties in each update of the payment channel. However, the required communication reduces the performance of the channel and the channel can only be updated if all parties are online. To manage a channel more efficiently, a central party can be introduced. There are proposals using a fixed [Pan+19] or a dynamically elected [Ye+21; Che+25] leader for that purpose. An alternative approach for building a multi-party payment channel is to build a virtual multi-party payment channel based on two-party payment channels [Che+22].

A special type of a multi-party channel is a channel factory [BDW17; PPT19]. A channel factory is a multi-party channel with the purpose of opening and closing two-party channels off-chain. Thereby, channel factories are a second layer on top of the blockchain and the two-party payment channels become a third layer.

2.4.2.4. Payment Channel Hubs

In several approaches for multi-party payment channels, there is a central party that facilitates maintaining the channel. If the parties in the multi-party payment channel open a channel directly to the central party, a payment channel hub is created. As the central party can observe all payments between the users connected to the hub, several works proposed approaches to protect the privacy of payments in payment channel hubs [Hei+17; TMM21; GM17; Qin+23]. For accountability of this central party and to allow for users to join and leave an open multi-party channel, the property of using the blockchain only for opening and closing the channel can also be given up. In commit chains [Kha+18], the central party periodically publishes a commitment to the current state on the blockchain. Similarly, in Magma [Ge+23] the blockchain is used as an arbiter in case the central party misbehaves or in case an update of the channel is made to include new users or exclude users from the channel. In Boros [Ye+19], the idea of a payment channel hub using commit chains is used for transfers between two-party channels which can be seen as a payment channel hub built on top of two-party channels. The idea of virtual channels is combined with payment channel hubs by PRI [Li+23] which is an approach for a payment channel

hub over which virtual two-party channels can be created with the distinct feature that users can create channels to the hub that are based on different first layer blockchains.

A further approach to increase trust into the central party in a payment channel hub, is to run the central party inside a trusted execution environment [GLH19; Erw+23; Lee+24]. Similarly, in Thunderbolt [Wen+25], the central party of the payment channel hub is replaced by a committee.

The problem of placing the payment channel hubs in a payment channel network as well as how to route payments in such a network is studied in [Yan+23].

2.4.3. Multi-Hop Payments

Before a multi-hop payment in a payment channel network can be executed, a route for the payment must be found. In the following, we first discuss several approaches for routing payments in payment channel networks. We discuss alternative approaches for executing a multi-hop payment that are different than the approach with HTLCs presented above in Section 2.3.5. By making payments in Lightning, channels can become imbalanced when a large share of a channel's capacity is owned by one party. We present approaches for rebalancing payment channels. Routing and rebalancing are important operational aspects of payment channel networks. While we do not contribute to these aspects in particular, we show in the following that other work has considered these aspects. Because this previous work makes practical deployments of payment channel networks more feasible, it strengthens the relevance of payment channel networks and, thus, of our work. Finally, we discuss privacy aspects of multi-hop payments. These privacy aspects are an example of security aspects beyond the security of user balances that we consider in the security property for payment channel networks in Chapter 3.

2.4.3.1. Routing

Lightning uses a source routing algorithm, i.e. the sender of a payment calculates a route for the payment through the network. This routing approach has the advantage that the sender has full control over the selection of the route and can consider own optimization criteria during route selection. However, the source routing approach requires each possible sender to know the topology of the payment channel network. Incomplete topology information might lead to non-optimal routes being chosen by the sender. Outdated topology information might lead the sender to choose an invalid route. Further, to determine whether a multi-hop payment can be executed, the sender has to know the balances inside each payment channel on the route. Making these channel balances public would improve the success rate of payments but have a severe impact on the privacy on users of the payment channel network because one could infer data about their behavior from the changes of channel balances [Tan+20]. Therefore, Lightning does not make channel balances public and uses source routing nevertheless. Routes chosen by a payment sender in Lightning do not take into account the balances on the route and, therefore, might fail

during payment execution. Consequently, sending a payment over Lightning is an iterative process of choosing a route for the payment followed by trying to execute the payment until a route has been found over which the payment could be executed. To improve on these properties of the source routing approach, most notably the limitation that payments can fail due to insufficient knowledge of the sender and the requirement that topology information must be shared in the entire network, several other approaches for routing payments have been proposed in the literature. Most routing algorithms proposed for payment channel networks are adaptions of routing approaches previously proposed for other networks. For example, the idea of landmark routing [Tsu88] has inspired several routing algorithms [Vis+12; Mal+17a; TB20; SK24] with the use of landmarks. In this context, a landmark is a "router whose neighbor routers within a certain number of hops contain routing entries for that router." [Tsu88]. Other approaches [Roo+17; ZSQ21] are based on embeddings [PR05]. In these approaches, each node is assigned an address which is constructed in such a way that a greedy routing strategy can be used so that with each forwarding of a message the message gets closer to the node that it has been sent to. Further, other routing algorithms [RLT17; Yu+18; Don+18; Wan+19b; Liu+23] are based on viewing the payment channel network as a flow network [JF56]. Recently, also reinforcement learning has been used as a component used in routing algorithms [KG21; VK25] for payment channel networks.

2.4.3.2. Multi-Path Payments

The success of a payment from one node to another node in a payment channel network depends on the distribution of balances of the channels on routes between the two nodes. It might be that several routes exist between two nodes but none of these routes has a sufficient capacity for the payment. To increase the success rate of payment attempts, several approaches [PN18; BNT20; Eck+20; MR23; DK23; Pic+21; NT24] have been proposed for protocols that split a payment into multiple smaller payments and aggregate them at the receiver.

2.4.3.3. Payment Execution

After a route has been found, the payment can be executed. In Lightning, payment execution is done in two phases: First, coins are locked using an HTLC in each payment channel on the route and secondly, the coins are released by fulfilling the HTLCs. Because the same hash is used in all HTLCs on a payment's route, an observer who has access to multiple nodes on the route can link HTLCs to the same payment. Several alternative locking mechanisms [Mal+17b; Mal+19; TM20; LA22; Zha+23; WRW24; Yu+19] have been proposed in which, for each payment channel on a payment's route, a different secret is used to increase privacy. The per-channel secrets are connected so that each forwarding hop can calculate the secret that the hop needs to fulfill the incoming HTLC based on the secret that the hop receives for the outgoing HTLC.

The use of HTLCs for multi-hop payments requires staggered timelocks along the payment route and, thus, the HTLC timelock depends on the length of the route. Other alternative locking mechanisms [Mil+19; EMM19; JLT21; Aum+21c; AAM22] reduce the length of the HTLC timelock by synchronizing all payment channels on a route to remove the need for a staggered timelock. In addition to that, some [Mil+19; EMM19; AAM22; Wan+24] of these locking mechanism are not restricted to linear paths but can be used for payments that include payments on unconnected payment channels. This property can be used to increase payment throughput by aggregating concurrent payments that use the same payment channels and 'cancel out' each other [Tiw+23].

2.4.3.4. Rebalancing

If a payment channel is used to send more funds into one direction that into the opposite direction, after a while, the entire capacity of the payment channel will be owned by one of the users. In this situation, the channel is unbalanced and can only be used to send payments into one direction. While the channel could be closed and opened with a new distribution of balances, such an on-chain rebalancing costs the fee for the two on-chain transactions to close and reopen the channel. Several approaches have been proposed for the problem of how to rebalance a payment channel without closing the payment channel. If a user has no funds in one channel but has funds in another channel, rebalancing can be done using a cyclic payment that equalizes the balances in the affected channels. After an approach [KG17] that uses a central party to coordinate a rebalancing between multiple parties, an approach [Ava+22] was proposed that preserves the privacy of the participating parties by hiding the balances inside the payment channels using multi-party computation. It was shown [PN20] that a rebalancing strategy using circular payments applied by each party in the network with only local knowledge can significantly improve the payment success rate in the payment channel network. Further work includes an approach [Hon+22] for automated asynchronous rebalancing on a regular basis instead of rebalancing on request and an approach [Ge+22] to construct payment channels so that capacity can be shifted from one channel of a user to another channel.

2.4.3.5. Privacy

Lightning tries to hide as much information as possible about payments that are conducted using Lightning. To this end, the balance of the two parties in a channel is hidden to the public (cf. Section 2.4.3.1) while the participating nodes and the capacity of the channel are public. It has been shown [Her+19; vDam+20; Tik+20; Kap+21; BNT22; RK22], however, that the balance inside a payment channel can be estimated by probing the balance with multiple payment requests and testing whether they are successful. A further privacy goal is that intermediate nodes who forward a payment should not learn the identity of the sender or the receiver of the payment. By exploiting knowledge about how Lightning clients construct multi-hop payments, it was shown [RT20; KER21] that intermediate nodes might be able to estimate the sender and the receiver of a payment.

2.4.4. Security and Management Aspects

In the following, we discuss several security and management aspects of payment channels. We start by discussing how watchtowers support users to deal with the requirement that users need to be regularly online to watch the blockchain for cheating attempts. In Chapter 6, we contribute two novel watchtower approaches that complement previous approaches by showing how watchtowers can benefit from the use of TEEs and IPFS [Ben14]. We further discuss the work related to the requirement of payment channels that a user must be able to timely publish a transaction when a cheating attempt is detected and, in particular, blockchain congestion and censorship that may result in this requirement not being met. We will again encounter this issue when we discuss whether Bitcoin fulfills the requirements of a payment channel network in Chapter 5 and find that Bitcoin does not guarantee that a user can always publish a transaction within a pre-specified time interval. We present griefing which is a denial of service attack that is possible in Lightning. This attack shows that there are attacks possible that are not directly related to user balances and, thus, do not break the security property that we define in Chapter 3. We further discuss research on how to optimize for fees that intermediate nodes earn for forwarding a payment. Finally, we present research related to the network graph of a payment channel network.

2.4.4.1. Watchtowers

Payment channel protocols require an honest user to regularly watch the blockchain and react to cheating attempts with a revocation transaction. In practice, this requirement might not be fulfillable if a user cannot ensure that the user's Lightning client runs continuously and has continuous network connection. To maintain security even when a user is offline, the task of watching the blockchain and reacting to cheating attempts can be delegated to a third-party, called a watchtower. Several approaches for protocols for watchtowers have been proposed with different focuses such as accountability [McC+19], incentives [Ava+18b; ATW20], storage [KNW19; Mir+22], availability [LSS20; XZZ24], and privacy [Mir+21]. In Sections 6.2 and 6.3, we present two further approaches for watchtowers that we have developed.

Beyond that, approaches have been explored for payment channels that modify or do not have the liveness requirement and, thus, do not need a watchtower. One approach [Ava+21] is to include into the payment channel protocol a committee of external parties that learns about each update of the payment channel and has to approve each closing attempt. Another approach [Aum+22; Yin+24] is to modify the liveness requirement to require liveness not continuously but only at predefined windows in time.

2.4.4.2. Blockchain Congestion and Censorship

The security of payment channels depends on the ability of publishing transactions on the blockchain in a timely manner. While this ability is given under normal circumstances,

there might be time windows during which many transactions are queued to be published on the blockchain. Such a time of congestion could be exploited by an adversary to steal funds of an honest user [HZ20]. This attack led to the development of mechanisms [Tra+22; Lot+22] that increase the resilience of payment channel protocols during times of blockchain congestion. A similar attack [RN21] shows that, if an adversary manages to control the network connections of a victim, the adversary might delay the delivery of new blocks to the victim so that the victim cannot punish cheating attempts within the given time.

A further reason why a user might not be able to publish a transaction is censorship by miners, i.e., miners do not include the transactions of the user in new blocks. If a user closes a payment channel dishonestly and offers a bribe to miners, it was shown [NKW21; Tsa+21; AT22; Ava+24] that it can be profitable for miners with large mining power to accept a bribe and censor the revocation transaction of the user who was cheated. However, the bribing attempt fails if there are enough miners who do not act according to the strategy that is game-theoretically optimal for them or if the parameters of the payment channel (fees and relative timelocks) are set to large enough values [NKW21; Tsa+21]. Also mechanisms [Tsa+21; AT22] have been proposed that extend payment channels to increase their resilience against bribery attacks, e.g., by disincentivizing bribery attempts by letting miners receive one or both user's balance if a user tries to bribe [Tsa+21; AT22].

2.4.4.3. Griefing

Multi-hop payments in a payment channel network can be mutually exclusive if they are routed in the same direction over the same channels and use in total more balance than available on this channel. This exclusivity can be used for a denial of service attack [Pér+20; KRK23] by making payments that deliberately lock a large share of the available balance in a channel for a long time and, thereby, prevent the channel from being used for forwarding other payments. If this attack is executed on multiple channels in a payment channel network at the same time, it can render the entire network unusable [MZ21a; TMM20; LHY22]. Lightning is susceptible to such an attack. A proposed countermeasure [MBR20] against this attack is to modify the multi-hop payment protocol so that the each hop in a multi-hop payment pays fees to the previous hop dependent on the processing time of the payment.

2.4.4.4. Fees

Payment channel networks rely on nodes that forward payments. Users are incentivized to run such nodes by fees that they receive for forwarding a payment. The operator of a node for relaying payments has to invest capital and time and receives fees as revenue. To maximize this revenue, the operator wants to optimally set the parameters for their node, in particular which channels to open and how to adjust the fees on these channels. This optimization problem has been studied [ERE20; BSB21; PT22; KER23]. From the

perspective of a payment's sender, it was also researched how to minimize the fees paid for a payment [Eng+17; ZYX19; BXW22].

2.4.4.5. Network Graph

Lightning is the largest deployed payment channel network and, to enable nodes to find routes for multi-hop transactions, the topology of the Lightning Network is public. In the following, we present results that were found about the actual topology of the Lightning Network and results that analyzed topology generation from a theoretical point of view.

The creation of payment channel networks was modeled in [BCV20; Ava+20] which led to the finding that a star graph topology would be game-theoretically optimal [Ava+18a; Ava+20]. This result might be an explanation why the empirical analyses observed an increase in centrality. A further work analyzed the theoretical achievable and the practical achievable success rate of payments [Tao+23]. A recent work showed that payment channel networks with a semi-hierarchical topology can be used in the architecture of a Central Bank Digital Currency [Ben+24; Ben+25].

Several works have researched how to optimize the topology of a payment channel network. There are approaches to generate a topology for a given set of nodes [Erd+21; KR21; KKR24] and to optimize the capacities in a given topology so that the success rate of payment's is increased [POR24]. Further there are approaches to optimize the placement of payment channel hubs which are nodes that are connected to many other nodes [WJ22; Yan+23]. The effect of attachment decisions of individual nodes on the network topology was analyzed in [LRT21].

2.4.5. Empirical Results

In this subsection, we present results of empirical studies that measured various metrics related to payment channel networks, in particular, the Lightning Network. In Chapter 5, we also contribute empirical measurements. However, our measurements consider the Bitcoin peer-to-peer network which is the layer underlying the Lightning Network. Thereby, our work complements the following results that focus primarily on the payment channel network itself.

2.4.5.1. Topology

The topology of the Lightning Network is public and information about the topology is shared between all nodes of the Lightning peer-to-peer network. Several works have analyzed the topology at different points in time. The main foci of these works were to analyze the centrality of the network and the susceptibility to failures of individual nodes in the network. Consistently, it was shown that while the Lightning Network withstands the removal of random nodes, the targeted removal of central nodes significantly reduces

the success probability for routing multi-hop payments [RMT19; CVD19; Ser+20; MF20]. Further, the graph was analyzed using different centrality metrics [OPH22] and it was shown that a small subset of nodes have a high centrality and the centrality has increased over time [KS20; Cam+22; Zab+22; Zab+24].

2.4.5.2. Payment Success Rate and User Behavior

Empirical studies beyond the topology of the network analyzed for example the success rate of payments. Waugh and Holz [WH20] found that, in 2019, the success rate for payments from a fixed node to random other nodes was only about 72 % for small payments of 0.01 USD and went down to 17 % for payments of values of 100 USD. They also found that, while there is low churn in nodes, the distributed topology information is often stale [WH20]. Another work analyzed the networks that nodes in the Lightning Network were running and found that most nodes are located in networks of ISP's which indicates that they are run privately by end-users or run in the Tor network [Cas+21]. By looking at how channels were closed, a further work found that more than half of channels are closed cooperatively and revocations are extremely rare [Grö+26]. Interestingly, 60% of cooperatively closed channels are used to fund another channel [Grö+26]. An analysis of HTLCs found in commitment transactions of uncooperatively closed channels revealed that HTLCs had an amount of on average 260 000 Sats and the payment values were often round numbers (e.g. 1000, 2000, 3000, or 10 000).

2.5. Conclusion

In this chapter, we have introduced the idea of payment channels and payment channel networks. We presented Lightning as an implementation of a protocol for a payment channel network and we presented Bitcoin as the underlying payment system. We showed that HTLC-based payments provide a fair exchange and we will use this property again in Chapter 5 for the development of a ticketing system for public transport based on a payment channel network. The chapter also gave an overview of research in the area of payment channels and situated the contributions of the following chapters within this broader research landscape.

3. Formalization of a Payment Channel Network

The protocol for a payment channel network combines a payment channel protocol as an implementation of the payment channel contract (see Section 2.1) with an implementation of a multi-hop payment protocol. In this and the following chapter, we consider the research question: How to verify that a payment channel network fulfills its security properties? We answer this question by describing and implementing an approach for such a verification. To this end, we formalize the Lightning protocol and we formalize a security property in the formal language TLA$^+$. The security property guarantees that an honest user will finally be paid out the user's correct balance. In this chapter, we describe our formalization of the Lightning protocol and the security property. In the following chapter, we use model checking to check that the specification conforms to the security property. While model checking is in principle a method that is highly automated, we cannot trivially apply model checking to our specification of Lightning because the state space of the specification is too large. We make model checking possible by using a stepwise refinement approach in which model checking is combined with pen-and-paper proofs. This verification approach is described in the following chapter.

The content of this and of the following chapter is based on the previous publications [GH22; GH26a].

In the first section of this chapter, we present related security analyses of Lightning and give an introduction into the formal language TLA$^+$ that we use for modeling the Lightning protocol and the security property. We continue by describing our formalization of the security property in Section 3.2 and our TLA$^+$ formalization of Lightning in Section 3.3.

3.1. Fundamentals

In this section, we first give an overview of previous security analyses of Lightning and of primitives used by Lightning. Then, we given an introduction into TLA$^+$ which is the formal language that we use to specify Lightning and the security property.

3.1.1. Related Work: Formalization and Security Analyses of Lightning

The Lightning protocol as well as similar protocols based on Bitcoin have already been analyzed previously. We start our summary of related work by briefly presenting work that analyzed subprotocols and primitives used by Lightning and continue by presenting security analyses of Lightning with a similar focus to our work.

Bitcoin contracts using timelocks were modeled and verified using the Uppaal model checker [LPY97] in a work [And+14] that predates the Lightning protocol. A few years later, Tamarin [Bas+22] was used to formally model and analyze HTLCs [BGW20]. An analysis of secrecy and authenticity properties of four subprotocols of Lightning was conducted [HS20; HS22] using ProVerif [BCC22]. A further analysis [Rai+23; Bru+23] of two subprotocols of Lightning had a focus on game-theoretic security. Further, the Spin model checker [Hol97] was used to analyze a subprotocol of Lightning [Wei+24] for single hop payments. Our work differs from these previous approaches to analyze security properties of Lightning by the scope of the analyzed protocol and by the analyzed properties.

The security of the Lightning protocol was analyzed with a similar scope to our work by Kiayias and Thyfronitis Litos [KT20]. They use the Universal Composability (UC) framework [Can01] to specify Lightning and an ideal functionality that describes the security property of Lightning. Using a pen-and-paper proof, they prove that their specification of Lightning implements the ideal functionality. The specification of Lightning in [KT20] closely models the cryptographic primitives used by Lightning. In contrast, we abstract cryptographic primitives for the formalization that we present in this chapter. We found that the specification of Lightning in [KT20] has two flaws that render the protocol insecure. The first flaw is that the specification of how a user reacts to outdated transactions does not handle all possible cases. The second flaw concerns how spending conditions in inputs and outputs of transactions are modeled. Because of simplifications introduced while modeling transactions, the specification allows transactions to be published in situations in which they should not be publishable leading to a violation of the security property. A more detailed description of the flaws can be found in [GH25, Appendix A]. These flaws are not flaws of Lightning but of the concrete specification of Lightning in [KT20]. While these flaws can be corrected, they show that solely a pen-and-paper proof does not give sufficient confidence that Lightning actually fulfills the analyzed security property. Consequently, it is preferable to use automated techniques such as model checking and theorem proving to verify Lightning.

Concurrently to our work, a simplified variant of Lightning that considers only single-hop payments was formalized by Fabiański et al. [FSL26] using the program verification platform Why3 [Bob+11]. As in this chapter, they also consider the informal security property that an honest user should not lose money. However, they use a more complex game-based definition in contrast to our approach of defining the security property by defining the behavior of a secure system. While they only consider a simplification of Lightning that does not include HTLCs and multi-hop payments, their approach has a

stronger adversary model and provides a machine checked proof. This work by Fabiański et al. shows that writing a verifiable proof even for a simplified variant of Lightning is a challenging effort.

3.1.2. TLA⁺

The Temporal Logic of Actions (TLA) [Lam94] was developed by Lamport to formally reason about the behavior of systems. TLA⁺ is a language for TLA to specify systems and properties. Because we use TLA⁺ to specify Lightning, we give a brief introduction to TLA⁺. For a complete definition of TLA⁺, we refer the reader to [Lam02].

In TLA⁺, a system S is defined by describing its behaviors. Formally, a system S is defined as the set of possible behaviors of the system. A *behavior* $\sigma = \langle s_0, s_1, s_2, ... \rangle \in S$ is defined to be an infinite sequence of states s_i. A state is defined to be an assignment of values to a set of variables v. We use the common notation $s.v$ to refer to the value of the variable v in state s. The state space of a system is the set of all states that occur in a behavior of the system. A *step* $s \rightarrow t$ is a pair of successive states s and t in a behavior. An *action* is a function that assigns a boolean value to a step. If an action is TRUE for a step $s \rightarrow t$, we say that the step is an A step and write $s \xrightarrow{A} t$. An action A is *enabled* in state s if there exists an A step starting in state s.

A temporal formula F in TLA⁺ assigns a boolean value to a behavior σ. As TLA is a temporal logic, TLA⁺ contains, additionally to the usual mathematical operators, operators for temporal reasoning: $\Box F$ defines that formula F is always, i.e. in all states of the behavior, valid. $\Diamond F$ is defined to mean that F is valid at least once, i.e, there is at least one state in the behavior in which F is valid. Further, TLA⁺ contains the prime operator $'$ that is used to define actions. Primed variables in a formula describe the value in the second state of a step while unprimed variables describe a variable's value in the first state of a step. If a formula F is primed, then F' means F with all variables in F primed.

To describe the behaviors of a system, a system is specified by defining the initial states of the system and how the state of the system can change. This description is done using a single formula in TLA⁺that is a specification of the system. Throughout this work, the terms system and specification are interchangeable in most cases. Typically, such a formula is written as $Spec = Init \wedge \Box[Next]_v \wedge Liveness$. The initial states of the system are defined by the unprimed part of the formula $Spec$ which is usually defined in the $Init$ formula. The notation $[Next]_v$ is an abbreviation for $Next \vee v' = v$. In a step that fulfills $v' = v$ the values of all variables are unchanged. Such a step is called a stuttering step. The action $Next$ is an action that describes all possible steps of the system. The formula $Liveness$ is a liveness condition that must be fulfilled by all behaviors of a system. In this work, we only use the liveness condition for weak fairness $WF_v(A)$ which asserts that, if an A step is continuously enabled, an A step must be taken.

Example 3.1. Figure 3.1 shows a behavior and an action of an exemplary system. The system has a single variable *Time* that has an integer value. Each state of the system is an assignment of a

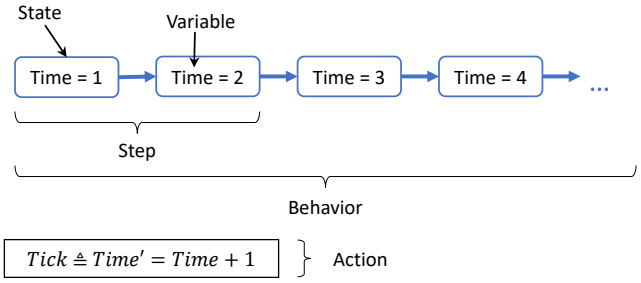

Figure 3.1.: Visualization of an example that shows how behaviors in TLA$^+$ are composed as well as an exemplary definition of an action.

value to the variable *Time*. In the shown behavior, the value of the variable *Time* is incremented in each state. A step is shown to be a pair of two successive states in the behavior. Figure 3.1 also shows the definition of an action *Tick*. The action *Tick* is true for the shown step in which the value of the variable *Time* is increased from 1 to 2 because the action specifies that the value of the variable *Time* in the second state of the step (which is 2) must equal the value of the variable *Time* in the first state of the step (which is 1) plus 1. As this condition is fulfilled for the shown step, the step is a step of action *Tick*.

The system shown in Fig. 3.1 could have the $Init$ predicate $Init \triangleq Time = 1$. However, the definition of $Init$ could also be $Init \triangleq Time \in Nat$ which states that the variable *Time* can start at any natural number. Then, other behaviors than the behavior shown would also be possible. The *Next* action of the system could be simply defined as $Next \triangleq Tick$.

In TLA$^+$, a function f assigns a value $f[x]$ to each value x in the function's domain. A record r in TLA$^+$ can have multiple fields that can be accessed using the notation $r.h$ for field h of record r. A record with n fields can be written as $[h_1 \mapsto e_1, ..., h_n \mapsto e_n]$ in which $r.h_i = e_i$ for $i \in \{1, ..., n\}$.

To define a system in TLA$^+$, one specifies custom operators which are comparable to the concept of functions in imperative programming languages. An operator *Op(a, b, ...)* is an expression that can have various parameters and must be defined in the specification. In a formula that contains the operator *Op*, the operator *Op* is replaced by the definition of the operator and all instances of the parameters $a, b, ...$ in the definition of the operator are replaced by the expressions passed to the operator.

3.1.2.1. Refinement

The variables of a system can be grouped into internal variables and external variables. While external variables describe what can be observed from a system from the outside, internal variables cannot be seen by an external observer. In TLA$^+$, it can be expressed that one system implements another system. A system S_1 implements (or, equally, refines) system S_2, if all external variables of system S_1 are also external variables of system S_2 and the way how the external variables of system S_1 are changed in every behavior of system S_1 also fulfills system S_2. This means that, for an external observer, the system

S_1 and the system S_2 are indistinguishable. Internally, the two systems might work very differently and have different sets of internal variables. If system S_1 implements S_2, we use the notation $S_1 \Rightarrow S_2$.

To show that system S_1 implements S_2, we specify a mapping from the state space of system S_1 to the state space of S_2. With a slight formal incorrectness, we also say that behaviors are mapped by the mapping whereby we mean that a behavior σ_1 of system S_1 is mapped to a behavior σ_2 of system S_2 by applying the mapping to all states in σ_1. The mapping is a refinement mapping if it maps every behavior σ_1 of system S_1 to a behavior σ_2 of system S_2 so that the values of the external variables in both behaviors are equal and σ_2 is a valid behavior of system S_2. If there exists a refinement mapping from system S_1 to system S_2, then system S_1 implements system S_2 [AL91].

3.1.2.2. Explicit Real-Time Specifications

Lightning depends on time to decide whether an HTLC's timelock has expired. Accordingly, a specification of Lightning must model time explicitly. Specifications with this characteristic are commonly called real-time specifications. Although dedicated languages for real-time modeling exist, such as Kronos [Boz+98], and Uppaal [LPY97]), we adopt TLA$^+$ which is a general-purpose specification language.

Real-time systems can be represented in TLA$^+$ using *explicit real-time specifications* [Lam05]. In such specifications, time is modeled by a set of clock variables. There must be at least one clock; however, there might also be multiple clocks. Because time is defined in Lightning by the height of the blockchain, we assume that all clocks have discrete values. The passage of time is modeled by an *AdvanceTime* action (also called *Tick* in the literature) which increases each clock by the same non-negative integer value and leaves all remaining variables unchanged.

This form of an explicit real-time specification closely resembles the model of a timed automaton[1] [AD90]. In Chapter 4, we describe how we apply results from research in the area of timed automata to the Lightning specification.

3.2. Formalization of Secure Payment Network

Our goal is to model check that Lightning securely manages the balances of users. In this section, we describe how we define this security property formally. We first present an informal security definition. Based on the informal definition, we explain the variables that are described by the security property and in which ways these variables should be allowed to be changed. These allowed changes are specified in the formal security definition that is given in Figs. 3.2 to 3.4. We explain the structure and the actions of this

[1] One difference is that timed automata are typically defined with real numbered values for clocks while the time in Lightning is the height of the blockchain which is always an integer.

formal definition and conclude the section by explaining how this security property is used to show that Lightning is secure.

The security property is informally defined by Definition 2. The definition uses the property 'honest' for users. A user is defined to be honest if the user behaves as specified by the used protocol which is, in our case, the specification of Lightning.

Definition 2 (informal security)**.** An honest user eventually receives a payout on the blockchain of at least the user's correct balance.

We start with a first attempt to formalize this property. By discussing the suitability of this attempt for Lightning, we introduce the ideas behind our actual definition of the security property. The first attempt is formalized as follows:

$$\Diamond\Box(\forall\, user \in Users : Honest[user] \implies BlockchainBalance[user] \geq CorrectBalance[user])$$

If this property is true for a system, then it holds for every behavior of the system that, starting at some point in the behavior ($\Diamond$), it holds for all future states ($\Box$) for every user that, if the user is honest, then the user's blockchain balance is greater than or equal to the user's correct balance. This property is very simple and implicitly concerns three variables: Whether a user is honest, the blockchain balance of a user, and a user's correct balance. We assume that the first variable about the honesty of a user is a static property defined for each user. In the context of Lightning, a user's blockchain balance is defined as the amounts of outputs that a user can solely spend, i.e. funds that can be spent by a user but not by anyone else. The third variable *CorrectBalance* is unclear at this point and needs to be defined. The correct balance that a user should have is determined by the user's initial balance plus the amounts of received payments minus the amounts of sent payments. Thus, the value of a user's correct balance depends on a user's payments and, specifically, the payments' amounts and states. Therefore, the definition of the security property must also include the payments that are sent and received by each user. We can define *CorrectBalance[user]* in the first attempt above as *InitialBalance[user]* + $Sum(ReceivedPayments[user]) - Sum(SentPayments[user])$.

The definition of a user's correct balance shows that the informal security property above is insufficient from a practical point of view: The property guarantees that a user eventually receives what eventually the correct balance is. However, the value of the correct balance should be determined and guaranteed already while a protocol is executed. For example, when a user receives a payment, the user should be able to see that the user's correct balance was increased by the payment's amount. To describe the ongoing behavior of the correct balance, it is not sufficient to define the security property as a property that holds eventually but, instead, the security property must be defined as an abstract secure payment system that defines how the users' balances may be updated. While a user should immediately see an update of the user's correct balance when the user receives a payment, it is only guaranteed eventually that the user's balance on the blockchain matches (or exceeds) the user's correct balance.

module $SpecificationV$

variables $BlockchainBalances$, $ChannelBalances$, $Payments$, $Honest$
constants $UserIds$, $InitialPayments$, $Numbers$

$IdealUser(user) \triangleq$ instance $IdealUser$ with
$UserId \leftarrow user$,
$ChannelBalance \leftarrow ChannelBalances[user]$,
$BlockchainBalance \leftarrow BlockchainBalances[user]$,
$Payments \leftarrow Payments[user]$,
$Honest \leftarrow Honest[user]$

$IdealPayments \triangleq$ instance $IdealPayments$

$Spec \triangleq$
$\land \forall\, user \in UserIds : IdealUser(user)!Spec$
$\land\ IdealPayments!Spec$

Figure 3.2.: *SpecificationV* is the definition of the security property. The specification is composed of an instance of the module *IdealUser* for each user of the specification and the module *IdealPayments*. The specification is fulfilled if the specifications for all users and of *IdealPayments* are fulfilled. First published in [GH26a]. Reproduced with permission from Springer Nature.

We define the security property as such a secure payment system in TLA$^+$ with four variables:

1. *Honest*: Whether a user is honest.
2. *ChannelBalance*: A user's balance in a channel which defines the correct balance that a user expects to retrieve eventually. This balance is determined by the initial deposit and subsequent payments.
3. *Payments*: A user's view on the state of the user's payments, i.e., whether a payment has been sent or received.
4. *BlockchainBalance*: A user's balance on the blockchain which are the funds that a user can directly access. The amount that a user finally withdraws is added to this balance.

By specifying actions that define how the state of the secure payment system may change, we define possible behaviors of a secure payment system and thereby define our security property for Lightning. The full specification of the security property is shown in Figs. 3.2 to 3.4. The definition of the security property is divided into three modules:

- The module *IdealUser* (see Fig. 3.3) defines three actions *Deposit*, *Pay*, and *Withdraw* that describe how the variables of single user can change.
- The module *IdealPayments* (see Fig. 3.4) ensures that the users' views on which payments have been processed are consistent.

module *IdealUser*

extends *Integers*, *SumAmounts*
variables *BlockchainBalance*, *ChannelBalance*, *Payments*, *Honest*
constants *UserIds*, *UserId*, *InitialPayments*, *Numbers*
assume $Numbers \subseteq Int$

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land\ BlockchainBalance \in Numbers \\
&\quad \land\ ChannelBalance = 0 \\
&\quad \land\ Payments = \{pmt \in InitialPayments : \lor\ pmt.sender = UserId \\
&\qquad\qquad\qquad\qquad \lor\ pmt.receiver = UserId\} \\
&\quad \land\ Payments \in \text{SUBSET}\ [amount : Numbers,\ id : Numbers,\ sender : UserIds, \\
&\qquad receiver : UserIds,\ state : \{\text{“NEW”}, \text{“ABORTED”}, \text{“PROCESSED”}\}] \\
&\quad \land\ Honest \in \{\text{TRUE}, \text{FALSE}\}
\end{aligned}
$$

$$
\begin{aligned}
&Deposit \triangleq \\
&\quad \land\ \exists\, amount \in 1 \,..\, BlockchainBalance : \\
&\qquad \land\ BlockchainBalance' = BlockchainBalance - amount \\
&\qquad \land\ ChannelBalance' = ChannelBalance + amount \\
&\qquad \land\ ChannelBalance' \in Numbers \\
&\quad \land\ \text{UNCHANGED}\ \langle Payments, Honest\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Pay \triangleq \\
&\quad \land\ \exists\, P \in \text{SUBSET}\ \{pmt \in Payments : pmt.state = \text{“NEW”}\} : \\
&\qquad \exists\, newState \in [P \to \{\text{“ABORTED”}, \text{“PROCESSED”}\}] : \\
&\qquad\quad \land\ Payments' = (Payments \setminus P) \cup \{[p\ \text{EXCEPT}\ !.state = newState[p]] : p \in P\} \\
&\qquad\quad \land\ \text{LET}\ processedPmts \triangleq \{p \in P : newState[p] = \text{“PROCESSED”}\} \\
&\qquad\qquad\qquad recvBal \triangleq SumAmounts(\{p \in processedPmts : p.receiver = UserId\}) \\
&\qquad\qquad\qquad sentBal \triangleq SumAmounts(\{p \in processedPmts : p.sender = UserId\}) \\
&\qquad\qquad \text{IN} \\
&\qquad\qquad\qquad \land\ ChannelBalance' = ChannelBalance + recvBal - sentBal \\
&\qquad\qquad\qquad \land\ ChannelBalance' \geq 0 \\
&\quad \land\ \text{UNCHANGED}\ \langle Honest, BlockchainBalance\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Withdraw \triangleq \\
&\quad \land\ BlockchainBalance' \in Numbers \\
&\quad \land\ BlockchainBalance' \geq BlockchainBalance \\
&\quad \land\ \exists\, amount \in 0 \,..\, ChannelBalance : \\
&\qquad \land\ ChannelBalance' = ChannelBalance - amount \\
&\qquad \land\ Honest \implies BlockchainBalance' \geq BlockchainBalance + amount \\
&\quad \land\ \text{UNCHANGED}\ \langle Payments, Honest\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Next \triangleq Deposit \lor Pay \lor Withdraw \\
&vars \triangleq \langle BlockchainBalance, ChannelBalance, Payments, Honest\rangle \\
&Spec \triangleq \land\ Init \land \Box[Next]_{vars} \\
&\qquad\ \land\ \mathrm{WF}_{vars}(ChannelBalance > 0 \land Honest \land Withdraw)
\end{aligned}
$$

Figure 3.3.: Specification of an ideal user in the security property. First published in [GH26a]. Reproduced with permission from Springer Nature.

module *IdealPayments*

```
extends Integers
variable Payments
constants UserIds, Numbers

Init ≜
    ∧ ∀ user ∈ UserIds :
        Payments[user] ∈ subset [amount : Numbers, id : Numbers,
            sender : UserIds, receiver : UserIds,
            state : {"NEW", "ABORTED", "PROCESSED"}]

Pay ≜
    ∧ ∀ user ∈ UserIds :
        ∨ ¬Honest[user]
        ∨ unchanged Payments[user]
        ∨ ∃ processedPmts ∈ subset {p ∈ Payments[user] : p.state = "NEW"} :
            ∧ ∃ newState ∈ [processedPmts → {"ABORTED", "PROCESSED"}] :
                ∧ ∀ pmt ∈ processedPmts :
                    (newState[pmt] = "PROCESSED" ∧ pmt.sender = user)
                        ⟹  ∨ ¬Honest[pmt.receiver]
                            ∨ ∃ rp ∈ Payments'[pmt.receiver] :
                                rp.id = pmt.id ∧ rp.state = "PROCESSED"
                ∧ Payments[user]' = (Payments[user] \ processedPmts)
                                    ∪ {[pmt except !.state = newState[pmt]] :
                                        pmt ∈ processedPmts}

Spec ≜ Init ∧ □[Pay]_Payments
```

Figure 3.4.: Specification of the module *IdealPayments* that specifies that a payment step is only possible if the receiver of the payment sees the payment already as `processed` or updates to this state in the same step. First published in [GH26a]. Reproduced with permission from Springer Nature.

- The module *SpecificationV*[2] (see Fig. 3.2) connects the two other modules by defining that for all users in the payment network the specification of the module *IdealUser* must hold and that the specification of the module *IdealPayments* must hold.

Initially, the variables are initialized (see Fig. 3.3) so that each user has a channel balance of 0, an integer balance on the blockchain, and a set of payments that the user wants to send or receive. The variable *Honest* specifies for each user whether the user is honest or dishonest.

In the following, we describe the actions of the modules *IdealUser* and *IdealPayments*. The action *Deposit* describes a deposit by defining that a user's blockchain balance may

[2] The name *SpecificationV* comes from the context of the stepwise refinement that we will present in Chapter 4 where the security property is the fifth specification in a series of abstractions.

decrease and the user's channel balance may increase by the same amount. The action *Withdraw* describes a full withdraw by defining that a user's channel balance may be set to 0 and, for an honest user, the user's blockchain balance must increase by at least the user's channel balance. The balance of an honest user may increase by more than the user's channel balance because it might be that the other user has cheated and the honest user has punished the other user and, therefore, the honest user has received also the balance of the other user. For dishonest users, it is only guaranteed that a dishonest user's blockchain balance does not decrease. The action *Pay* of the module *IdealUser* describes the execution of a set of payments. The action *Pay* defines that the sending users' channel balances are decreased by the amounts of sent payments and the receiving user's channel balances are increased by the respective amounts. Payments may also be aborted, leaving channel balances unchanged.

A further aspect that is not captured by the informal definition of security in Definition 2 but by the formal definition of the security property, is that users intuitively expect from a secure payment system that the sender of a payment sees the payment as sent and the payment's balance is deducted from the sender's channel balance only if the receiver of the payment sees the payment as received. This consistency is enforced by the action *Pay* in the module *IdealPayments*. The action *Pay* defines that a payment can only be sent if the receiver of the payment has already received the payment or receives the payment in the same step. Further, the module *IdealUser* defines a fairness condition ensuring that the system does not terminate before all users have been paid out. The 'eventually' aspect of the security property is modeled by the action *Withdraw* together with the fairness condition: The variable *BlockchainBalance* is only increased during a step of the action *Withdraw* which can occur an arbitrary time after payments have been made. However, the fairness condition ensures that a *Withdraw* step occurs eventually and users actually receive their expected balance on the blockchain.

The definition of the secure payment system also implicitly defines that a processed payment cannot be undone: There is no action that changes the state of a processed payment. This property has the practical relevance that the seller of an item who expects a payment from a buyer knows that, if the payment has been received, the state of the payment is final and the item can be transferred from the seller to the buyer.

In the next chapter, we describe how we verify that Lightning implements the security property as defined above. In our formalization of Lightning, the *BlockchainBalances* variable is refined as the sum of unspent transaction outputs on the blockchain that a user can exclusively spend. From a user's perspective, the state of a payment changes from `new` to `processed` when the corresponding HTLC is fulfilled. Because we verify that using Lightning with this definition of a processed payment fulfills the security property, this shows how Lightning can securely be used: Honest users who sell a product in exchange for a payment can securely deliver the product once they have fulfilled the corresponding incoming HTLC and do not need to wait for the fulfilled HTLC to be removed from the channel.

The result that the protocol specification implements the secure payment system means that all modeled behavior cannot break the security property. Because our specification of

the protocol also models adversarial behavior this also holds for the modeled adversarial behavior and shows that the countermeasures implemented in the protocol are sufficient.

3.3. Formalization of Lightning

We choose TLA$^+$ to model Lightning because, as a general-purpose language, TLA$^+$ allows for specifying arbitrary protocols and because there are tools supporting different verification methods, e.g. random state space exploration (simulation), explicit-state model checking [Var25b], symbolic model checking [Var25a], and theorem proving [Cen22]. We compare the choice of TLA$^+$ to other approaches in the following chapter in Section 4.8.1.

To verify that Lightning fulfills the security property, we need to formalize Lightning because there exists only an informal official specification. In this section, we explain important aspects of our TLA$^+$ formalization of Lightning to show the assumptions and the abstraction layer of the formalization. The complete TLA$^+$ formalization is available on Zenodo [GH26b].

There exists an official specification of Lightning [Lig26c], documented in a series of so called BOLTs (Basis of Lightning Technology). This formalization is informal and there is no straightforward process for formalizing this specification. The main purpose of the BOLTs is to facilitate interoperability between different implementations of Lightning. Thus, the specification focuses mostly on specifying the communication between clients running the Lightning protocol, e.g. by specifying the messages that are exchanged, what content these messages may have, and how a client may react to messages. However, the BOLTs do not specify how a client internally stores its state which is an aspect that a formalization of Lightning in TLA$^+$ needs to consider. Because the formalization of Lightning can be seen as an implementation of Lightning in TLA$^+$, we have to make several design decisions that are not specified by the BOLTs.

Intuitively spoken, the key idea of a payment channel protocol is to send, forward, and receive payments and to ensure that the correct state of each payment can be enforced on the blockchain by ensuring that *each user can spend the right transaction output at the right time*. Thus, we include everything in the formalization that is required to specify how payments are sent, forwarded, and received and how an output can be spent. To model how an output can be spent, we model, for example, the keys exchanged by users, signatures for transactions, and preimages for HTLCs. We try to model Lightning in TLA$^+$ in a way that is close to an actual implementation. Therefore, we model message exchanges as specified in the BOLTs and we specify the behavior of each user based solely on information that the user has. To facilitate verification of the formalized protocol, we keep the formalization of Lightning as small as possible. Therefore, we do not model aspects of Lightning such as fees, key rotation for privacy protection, and mechanisms to ensure that a blockchain transaction pays enough fees to be inserted into the blockchain and we abstract cryptographic primitives like signatures and hash functions.

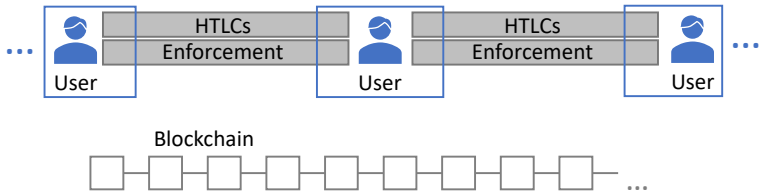

Figure 3.5.: Overview of the aspects modeled in the specification. The specification models users and the blockchain as actors. Users can send messages to each other and there can be payment channels between pairs of two users. If two users are part of the same payment channel, they communicate on two layers: The HTLC layer to perform multi-hop payments by exchanging payment information to add HTLCs and preimages to fulfill HTLCs. The enforcement layer to manage the payment channel by exchanging signatures and secrets to maintain the ability to enforce the state of an HTLC on the blockchain by closing the payment channel.

The TLA$^+$ formalization specifies a system with an arbitrary number of users. The behavior of a user is specified as in the official specification [Lig26c] with some simplifications that we detail below. The specification models all steps that are required for a functional protocol to open, update, and close a payment channel as well as to make multi-hop payments. The specification also models adversarial behavior that could affect the ownership of outputs on the blockchain: A dishonest user might publish outdated commitment transactions and create and publish a transaction that spends outputs for which the dishonest user knows the required secrets. The specification also specifies how other users react to such adversarial behavior by publishing revocation transactions.

On a high-level, the system we specify with the TLA$^+$ formalization is shown in Fig. 3.5. The actors in this system are the users and the blockchain. While the blockchain has only a single action to create new blocks, users can take many different actions. Users are connected to other users via payment channels. We separated the logic of each payment channel into two layers: On the HTLC layer, users take actions that are required to process payments by creating, adding, and fulfilling HTLCs. On the enforcement layer, users take actions to send and receive messages that are necessary to enforce the state of an HTLC on the blockchain. This includes the steps to open, update, and close a payment channel. Users exchange messages with each other and interact with the blockchain by publishing transactions and observing transactions that are published by other users.

This section is organized as follows: We start by giving an overview of the structure of the TLA$^+$ specification in Section 3.3.1. We briefly present the modules used in the specification and how they interact. In Section 3.3.2, we explain the modeling ideas behind the formalization. E.g., how we model cryptographic primitives, messages, and time. We describe in Section 3.3.3 in more detail the subprotocol of Lightning for opening, updating, and closing a channel and present the actions used for formalizing this subprotocol in the module *PaymentChannelUser*. We continue by describing in Section 3.3.4 the subprotocol of Lightning for multi-hop payments and presenting the actions used for specifying multi-hop payments which are defined in the module *HTLCUser*. We present further modeling aspects in Section 3.3.5 and discuss the relation of the specification of Lightning to the security property in Section 3.3.6. We conclude in Section 3.3.7.

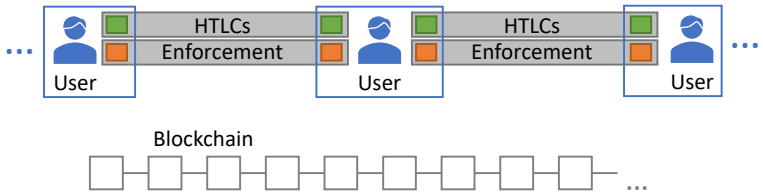

Figure 3.6.: Refined version of the system overview in Fig. 3.5. For each payment channel, there is an instance of the module *HTLCUser* (green boxes) on the HTLC layer for each user and an instance of the module *PaymentChannelUser* (orange boxes) on the enforcement layer for each user. Each step of the system is a step of one the instances of the modules *HTLCUser* or *PaymentChannelUser* or the blockchain (module *LedgerTime*).

In this section, we alternately present the Lightning protocol according to the BOLTs in more detail and explain how we formalized the protocol in TLA$^+$. To distinguish explanations of Lightning from descriptions of the formalization, explanations of Lightning are indented and marked with a yellow line on the left.

The formalization considers a set of users and a set of channels between these users. In this and the following chapter, we use U to denote the set of users and C to denote the set of channels. In the TLA$^+$ specifications, these sets are named *UserIDs* and *ChannelIDs* respectively. We usually use u to refer to a user $u \in U$ and c to refer to a channel $c \in C$. We use U_c to refer to the users that are part of channel $c \in C$.

3.3.1. Structure of Specification

The specification of Lightning is structured into several modules. The two main modules are the modules *PaymentChannelUser* and *HTLCUser*. The behavior of each user is modeled by two instances of the two modules *PaymentChannelUser* and *HTLCUser* (see Fig. 3.6). The module *PaymentChannelUser* specifies actions on the enforcement layer, i.e. actions related to managing transactions that can be published on the blockchain and the module *HTLCUser* specifies actions on the HTLC layer which are actions related to sending, receiving, and forwarding payments. In the following, we introduce these modules in more detail.

In this section, we show several excerpts of TLA$^+$ code. To focus the presentation, we shorten and simplify the code examples. For example, we omit the UNCHANGED conjuncts that list the variables that are not changed by an action.

The main module for the Lightning specification is called *SpecificationI*. The Roman numeral I as suffix indicates that we will later build other specifications based upon the Lightning specification (see Section 3.3.1.4).

In the following, we first introduce the variables of *SpecificationI* in Section 3.3.1.1. Then, we give an overview of the steps of *SpecificationI* in Section 3.3.1.2. In Section 3.3.1.3, we present how communication between modules is modeled in *SpecificationI*. Finally,

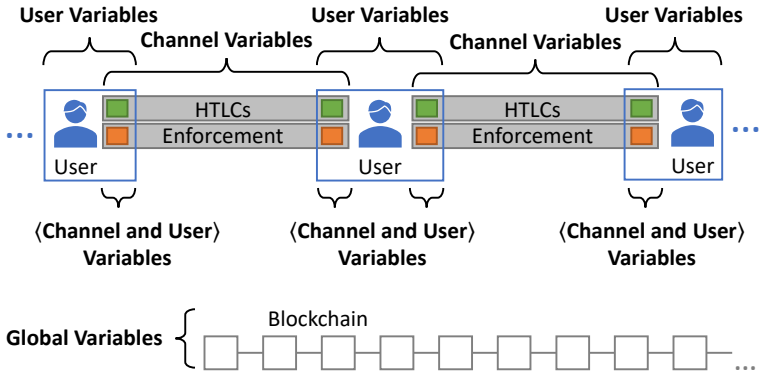

Figure 3.7.: Overview of the variable types used in the TLA$^+$ specification of Lightning and their scopes. ⟨Channel and user⟩ variables are specific to a channel and specific to a user. A set of these variables exists for each channel for each of the channel's users. User variables are specific to a user. They can be accessed by all of the user's instances of the module *PaymentChannelUser* and *HTLCUser*. Channel variables are specific to a channel. They can be accessed by the instances of the modules *PaymentChannelUser* and *HTLCUser* of both users of the channel. Global variables exist only once and can be accessed from any instance of any module.

we give in Section 3.3.1.4 a brief overview of the abstractions of *SpecificationI* that are presented in the following chapter.

3.3.1.1. Variables of *SpecificationI*

SpecificationI uses several variables to represent the state of the system. Here, we give an overview of these variables. In the remainder of this chapter, the variables will be described in more detail within their context. In the appendix in Section A.2, we give a brief overview that lists each variable and their purpose. There are four groups of variables (see Fig. 3.7): Global variables, user variables that are specific to a user, channel variables that are specific to a channel, and ⟨channel and user⟩ variables that are specific to a pair of channel and user. We use angle brackets to identify the last group of variables to avoid confusion between user variables, channel variables, and ⟨channel and user⟩ variables.

Global variables exist once in the specification. User variables exist once for every user. We model such a set of one variable per user with a function that assigns to each user identifier the respective variable. E.g., the variable *UserPayments* is a function and *UserPayments[u]* for a user u is the variable that contains payment records of user u. The names of all user variables start with the prefix *User*. Analogously, channel variables exist once for every channel and ⟨channel and user⟩ variables exist once for every pair of channel and user and are prefixed by *Channel* and *ChannelUser* respectively.

Figure 3.8 gives an overview of most variables of the specification grouped by their type. Global variables are used to model the exchange of messages between users and to model the blockchain. The blockchain is modeled with three variables that model the transactions included in the blockchain (*LedgerTx*), the current height of the blockchain (*LedgerTime*), and the age of each transaction in the blockchain (*TxAge*). The global variable *Messages*

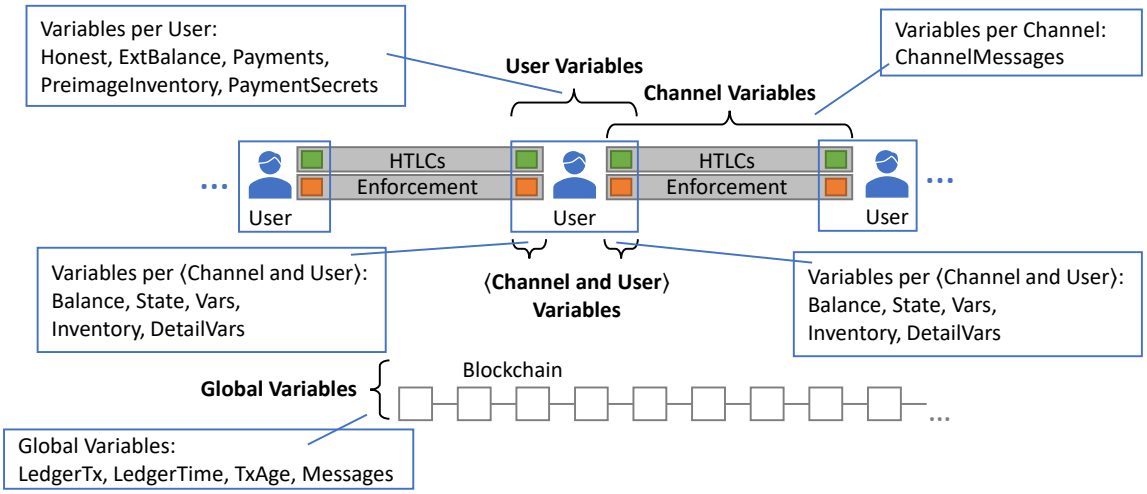

Figure 3.8.: Examples for different types of variables in the context of Fig. 3.6. The variable names are shortened by omitting the prefixes *User*, *Channel*, and *ChannelUser*.

models messages that users exchange for requesting and delivering invoices that contain the necessary data to initiate a payment. User variables are used to model whether a user is honest, the external balance of a user (*UserExtBalance*), a user's set of payments (*UserPayments*), and the preimages and payment secrets known by a user (*UserPreimageInventory* and *UserPaymentSecrets*). The ⟨channel and user⟩ variables, that exist once for every channel that a user has, model the user's balance in the channel (*ChannelUserBalance*), the user's protocol state (*ChannelUserState*), an inventory that stores keys and signed transactions (*ChannelUserInventory*), and two variables that are records that contain further variables (*ChannelUserVars* and *ChannelUserDetailVars*). The variable *ChannelUserVars*, for example, contains two sets *Vars.IncomingHTLCs* and *Vars.OutgoingHTLCs* that contain records for each incoming and outgoing HTLC. The channel variable *ChannelMessages* is specific to a channel and shared between the two users of the channel. This variable is used for message exchange between the two users within the channel (see Section 3.3.1.3).

An aim for the specification is to be close to an actual implementation. Therefore, the behavior of users is modeled so that each users accesses only the variables of the respective user. Communication between users is explicitly modeled using messages. In principle, a TLA$^+$ specification could model the behavior of a payment channel network in a different way by using a global view and changing the variables of multiple users in the same step or by using knowledge of the variables of one user to change the variables of another user. While the specification of Lightning does not make use of this global view, the specifications in the stepwise refinement that is presented in Chapter 4 use an increasingly global view as the level of abstraction increases.

3.3.1.2. Steps of *SpecificationI*

The main module of *SpecificationI* uses definitions from several other modules to define the specification. The most relevant modules are:

$$
\begin{aligned}
&NextI \triangleq \\
&\quad \lor\ AdvanceLedgerTimeI \\
&\quad \lor\ \exists\, c \in ChannelIDs : \exists\, u \in UsersOfChannelSet(c) : \\
&\qquad\qquad PaymentChannelUser(c,\ u)!Next \\
&\quad \lor\ \exists\, c \in ChannelIDs : \exists\, u \in UsersOfChannelSet(c) : \\
&\qquad\qquad HTLCUser(c,\ u)!Next \\
&\quad \lor\ WithdrawBalancesAfterChannelClosed
\end{aligned}
$$

Figure 3.9.: *NextI* action of *SpecificationI*. Each step of the action is either a step that advances time, a step to withdraw all balances after all channels have been closed, or a step of either the module *PaymentChannelUser* or *HTLCUser* of one of the two users in one of the channels.

- The module *PaymentChannelUser* contains the actions of a user with regard to the management of a payment channel. The module *PaymentChannelUser* defines the possible actions of a specific user in a specific channel. These actions describe the steps required for opening the channel, updating the channel, and closing the channel on the blockchain. The module *PaymentChannelUser* is instantiated once for every user and channel (see orange boxes in Fig. 3.6).
- The module *HTLCUser* contains the actions of a user with regard to HTLCs. The module *HTLCUser* defines the possible actions of a specific user in a specific channel for processing payments. These actions include the request for and sending of an invoice to initiate a payment, for adding HTLCs, and for resolving HTLCs either by fulfilling them or by timing them out. The module *HTLCUser* is instantiated once for every user and channel (see green boxes in Fig. 3.6).
- The module *LedgerTime* defines the actions of the blockchain, in particular, how time is advanced by creating new blocks.

Additionally, there are a few helper modules that contain operators that are used by the other modules.

Each step of *SpecificationI* is a step of the action *NextI* as defined in Fig. 3.9. This definition of *NextI* defines each step to be a step that either advances the time, a step in the module *PaymentChannelUser* for one user in one channel, a step in the module *HTLCUser* for one user in one channel, or a step to finally withdraw the users' balances after all channels have been closed (*WithdrawBalancesAfterChannelClosed*). The final withdraw step implements the action *Withdraw* in the security property (Fig. 3.3) by setting a user's channel balance to 0 and by setting a user's blockchain balance to the sum of amounts of outputs that the user can spend. The modules *PaymentChannelUser* and *HTLCUser* are parameterized by a channel and a user. We describe the actions in the module *PaymentChannelUser* in Section 3.3.3 and the actions of the module *HTLCUser* in Section 3.3.4.

3.3.1.3. Communication

The actions of *SpecificationI* are distributed over the modules *PaymentChannelUser*, *HTLCUser*, and *LedgerTime*. Each of these modules is defined so that it uses only local knowl-

edge: Each instance of the modules *PaymentChannelUser* and *HTLCUser* is parameterized by a user u and a channel c. The actions of these modules are defined so that only the user variables of user u, the channel variables of channel c, and the corresponding ⟨channel and user⟩ variables are accessed. To allow for exchange of information between the instances, the following communication paradigms are used:

Same user, same channel The instance of *PaymentChannelUser* and the instance of *HTLCUser* for the same user u and same channel c communicate via shared variables; primarily via the shared ⟨channel and user⟩ variables but also via shared user variables. Example: The state of an HTLC is stored in the variable *ChannelUserVars[c][u]* that is accessed by the modules *PaymentChannelUser* and *HTLCUser*.

Same user, different channel The instances of the modules *PaymentChannelUser* and *HTLCUser* for the same user c and different channels communicate via the shared user variables. Example: A preimage learned on one channel is added by an instance of *HTLCUser* to the variable *UserPreimageInventory[u]* and can be accessed by an instance of *HTLCUser* for another channel of the same user u.

Different user, same channel Two instances of the module *PaymentChannelUser* and two instances of the module *HTLCUser* of the same channel c communicate via message passing. Because TLA$^+$ does not natively support message passing, message passing is simulated using the shared channel variable *ChannelMessages[c]*. In the variable *ChannelMessages[c]*, in-transit messages are stored and this variable can be accessed by both instances of the same channel c.

Different user, different channel If a user wants to send a multi-hop payment to another user, the payment's sender requests an invoice from the payment's receiver. To this end, users might have to communicate although they do not share a payment channel. For the exchange of the request for an invoice and the invoice itself, also message passing is used. A global variable *Messages* stores all in-transit messages and can be accessed by any instance of *HTLCUser*.

The blockchain modeled by the variable *LedgerTx* also serves as an indirect communication channel between users of the same channel because users observe transactions on the blockchain that are published by the other user. This does not hold for users that do not share a payment channel because we assume that users do not monitor all transactions on the blockchain but only transactions that are relevant for their own channels.

3.3.1.4. Further Specifications

While *SpecificationI* is our formalization of Lightning, to facilitate model checking, we will introduce abstractions of this specification in Chapter 4. These abstractions are

- *SpecificationII* in which the protocol for managing a payment channel (the enforcement layer) is abstracted,
- *SpecificationIII* in which each whole payment channel is abstracted,

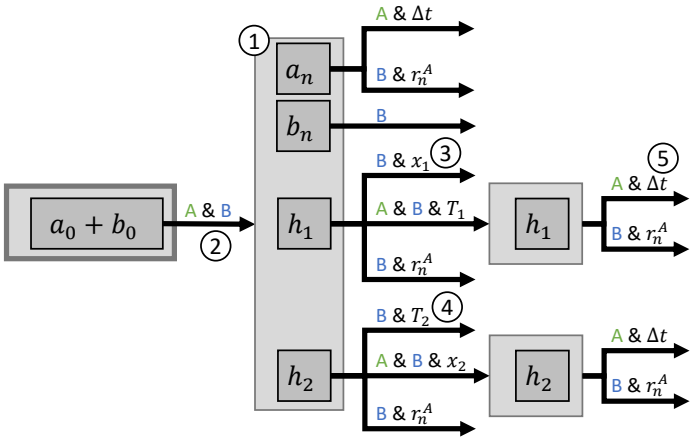

Figure 3.10.: Transaction tree showing a funding transaction, a commitment transaction with an incoming and an outgoing HTLC and two HTLC transactions.

- *SpecificationIV* in which the communication between payment channels is abstracted, and
- *SpecificationV* which equals the security property.

To distinguish the $Init$ predicates and the $Next$ actions of these specifications, we number these as well from $InitI$ and $NextI$ to $InitV$ to $NextV$.

3.3.2. Modeling Ideas

Before we present the actions of the specification in more detail in Section 3.3.3, we explain general ideas used in the TLA$^+$ specification. We start by presenting the formalization of messages, and continue with the modeling of verification and failure handling. Then we discuss different aspects of modeling transactions that include HTLCs (see Fig. 3.10, an adapted version of Fig. 2.8). For modeling transactions, we have to model the transaction structure (see Fig. 3.10, ①), cryptographic keys and signatures (see Fig. 3.10, ②), hash functions (see Fig. 3.10, ③), and absolute and relative timelocks (see Fig. 3.10, ④ and ⑤). Further we discuss how we model HTLC states and onion routing of messages during multi-hop payments.

3.3.2.1. Formalization of Messages

In general, we model that messages between users are delivered reliably but can be arbitrarily delayed. Our communication model differentiates between messages exchanged between the users of a channel and messages exchanged between users for multi-hop payments. The messages exchanged between users of a channel (actions of module *PaymentChannelUser*) are modeled to be delivered in order. The messages that users exchange for multi-hop payments are modeled to be received in arbitrary order.

The exchange of messages in a channel c is modeled by letting the users write messages to a message queue *ChannelMessages[c]* from which each user can read the first message for which the user is the receiver. In the formalization, we define the operator *SendMessage* that models the sending of a message by specifying that the channel's message queue *ChannelMessages[c]* is extended by the message that is sent. A message contains a field for the recipient, the sender, the message's type, and the payload. For each type of message, the formalization includes an action that sends the message and an action that expects such a message and reacts to the message. An action of user u that reads a message contains a condition that checks that the first message sent to user u has the expected type. Such an action that receives a message updates the receiving user's state based on the message's content and removes the message from the queue *ChannelMessages[c]*.

When starting a multi-hop payment, the payment's sender requests an invoice from the payment's receiver and the payment's receiver replies with an invoice that contains the preimage that should be used for the payment. We model this message exchange that is not related to a specific channel with the global variable *Messages*. The global variable *Messages* is a set that contains all messages for requesting and sending an invoice. Messages can be directly received or arbitrarily delayed because after a step in which a message was sent there can be no or an unlimited number of steps that advance time until a step is taken in which the message is received.

3.3.2.2. Verification and Failure Handling

> In Lightning, peers verify the contents of messages that they receive. This verification includes, for example, consistency checks and signature validations. If a verification fails, in most cases a peer should either send a `warning` message and close the communication channel or send an `error` message and close the payment channel on-chain. While the `warning` message facilitates debugging and can handle transient failures, the second case of sending an `error` message and closing the payment channel on-chain is the only option left in case of persistent failures.

In the formalization, we do not model the first case in which a `warning` message is sent because we assume that the communication channel does not cause transmission failures and, thus, resending a message would lead to the same verification result. We also do not explicitly model the sending of `error` messages because, while these messages help for debugging, they are not necessary for the security property of Lightning that we verify. The formalization models that received messages are verified. A failed verification is handled by ignoring the message and closing the channel. In the formalization, a payment channel can always be closed after the channel has been opened. Thus, at every state there exists for each party one behavior in which the party closes the channel by publishing their latest commitment transaction on the blockchain.

3.3.2.3. Hash Functions

Lightning relies on cryptographic primitives such as digital signatures and hash functions for which TLA$^+$ does not provide native models. In our formalization, we therefore adopt the standard perfect cryptography assumption, namely that cryptographic primitives cannot be broken by an adversary. Cryptographic keys, signatures, and hash values are represented symbolically as done in previous work (e.g., [And+14; BGW20]). Rather than attempting to model these primitives in full detail, we abstract them by capturing only the properties relevant to the Lightning protocol. In particular, we assume that a hash function is deterministic, efficiently computable given its preimage, infeasible to invert given only the hash value, and collision-resistant, i.e., distinct inputs produce distinct outputs. While Lightning specifies that preimages used in multi-hop payments are generated randomly, in the TLA$^+$ formalization, preimages are deterministically derived from the identifier of the corresponding payment ensuring that each payment is associated with a unique preimage. Furthermore, we model the hash of a preimage as being equal to the preimage itself and distinguish between preimages and hash values solely by the variables in which they are stored. For example, integers stored in the set *UserPreimageInventory[u]* represent preimages whereas an integer sent in an `update_add_htlc` message represents a hash value.

In Bitcoin, identifiers of transactions are generated using a hash function. In the formalization, we model transaction identifiers by drawing a new unique value for each transaction when the transaction is created and including that identifier in the transaction (see Section 3.3.2.5). The set *ChannelUsedTransactionIds[c]* is used to store the transaction ids that have already been used to ensure that each id is used only once. This approach also ensures that each transaction has a unique identifier and that the identifier for a given transaction can be easily obtained from the transaction.

3.3.2.4. Cryptographic Keys

In this subsection, we present the cryptographic keys that are used in Lightning and what they are used for. Lightning uses multiple asymmetric key pairs per user for different purposes. Because TLA$^+$ does not contain a built-in model of asymmetric key pairs, we use a symbolic representation of private and public keys. In particular, we use a record as a constant symbolic identifier per key pair. The same identifier is used for the private and the public key. Analogously to the model of hash functions, it can be seen from the context, e.g., from the field that an identifier is stored in, whether an identifier is a public or a private key.

> Lightning uses different key pairs for different purposes. The public keys of both users used to lock the output of the funding transaction are different from the public keys used in the user's outputs in a commitment transaction. The keys in a user's output in a commitment transaction are different depending on whether the commitment

> transaction has been created by the user themselves or by the other user. Still other keys are used in the conditions of HTLC outputs. The main reason for using different keys is to minimize the privacy leakage when some transactions are shared with other parties, e.g. when using a watchtower.

Because we focus our analysis of Lightning on balance security and not on privacy properties, we model all those mentioned key pairs with a single key pair per user.

> To incentivize honest behavior, the outputs of outdated commitment transactions that can be published by one party are made spendable for the other party. In Lightning, this mechanism is implemented using a dedicated revocation key pair per commitment transaction. Each public revocation key can be calculated by both parties. To calculate a private revocation key, secrets from both parties are required. To revoke a commitment transaction, one party sends their secret to calculate the private revocation key to the other party so that the other party can build the private revocation key. Thereby, only one party has the revocation secret and can revoke the commitment transaction. To allow for revoking each commitment transaction selectively, a different secret is used for every commitment transaction.

For the formalization, two aspects need to be considered: First, a different revocation key pair needs to be used for every commitment transaction. Second, an output in an outdated commitment transaction must be spendable only by a user who has both secrets required to calculate the private revocation key.

We model revocation keys as simple asymmetric key pairs that are modeled as records. Each of these records contains an identifier and a counter. To create a new revocation key pair, the counter of the previous revocation key pair is incremented by one. This approach achieves that there is a different revocation key pair for each commitment transaction and it can easily be seen during debugging what a key's purpose is. The second aspect requires that the outputs of a commitment transaction that can be published by one party can be revoked only by the other party. Because we model revocation keys as asymmetric keys, after one user has shared a private revocation key with the other user both users have the private key. To enable only one user to use the key for revoking a commitment transaction, we modify the spending conditions in commitment transactions: An output can only be revoked by a transaction that is signed using the private revocation key and the private key of the user who did not publish the transaction.

> Besides the use of new revocation keys for every commitment transactions, also the other keys are rotated in Lightning. If multiple commitment transactions of the same channel are shared with a third party, this approach makes it more difficult for the third party to link these commitment transactions.

We do not model this key rotation in the formalization because the purpose of the key rotation is privacy and the key rotation is not required for the balance security property that we focus on.

3.3.2.5. Transactions

Transactions are a crucial part of the formalization because they encode the logic which funds are spendable by which party. Thus, to analyze the balance security of Lightning, it is important to accurately model how transactions are built.

While we have introduced the types of transactions used in Lightning already in Section 2.3, we now explain these transactions in more detail and explain how their outputs are spendable.

The funding transaction contains one or more inputs that spend outputs owned by the funder and contains an output whose amount equals the capacity of the payment channel and which can be spent only by transactions signed by both users.

A commitment transaction has one input that references the funding transaction's output. Because spending this output requires signatures by both users, the commitment transaction must be signed by both users to be published on the blockchain. The outputs of a commitment transaction are as follows. We refer to the party who can publish the commitment transaction as the local party and to the other party as the remote party. Because we model the different key pairs of a user which are not used for revocation as a single key pair, we do not distinguish in the following description the different key pairs used in Lightning. We use parentheses in the following to clarify the precedence of the logical conditions described in natural language.

- The first output of a commitment transaction contains the funds of the local party. The output can be spent using a signature of the local party after a relative timelock of length *to_self_delay* or it can be spent using the private revocation key.
- The second output of a commitment transaction contains funds of the remote party and can be spent using the remote party's signature.
- Outgoing HTLCs: For each outgoing HTLC, there exists an output that can be spent either using the private revocation key or (using the remote party's private key and (either the local party's key or by providing the HTLC's preimage)).
- Incoming HTLCs: For each incoming HTLC, there exists an output that can be spent either using the private revocation key or (using the remote party's private key and ((either the HTLC's preimage and the local party's private key) or after the HTLC's timelock)).

A further type of transactions are HTLC success and timeout transactions that are created for a commitment transaction that contains HTLCs. These transactions may contain an absolute timelock called locktime. The HTLC success transaction has a locktime value of 0 which means that it is immediately valid. The HTLC timeout transaction has a locktime value of the HTLC's timelock, i.e. it is valid only after the point in time specified by the HTLC's timelock. An HTLC transaction's input references the respective HTLC output of a commitment transaction. The HTLC transaction must

> be signed using both parties' private keys. For an HTLC success transaction, the input must contain the HTLC's preimage. The output of both HTLC transactions is spendable either by the private revocation key for the associated commitment transaction or after *to_self_delay* using the local party's private key.

The formalization of transactions follows the UTXO (unspent transaction output) model of Bitcoin: A transaction is a record that contains a set of inputs, a set of outputs and additional data like the transaction's id, and an optional absolute timelock. An output of a transaction is a record that contains an id, an amount, and a set of conditions of which one needs to be fulfilled to spend the output. A condition consists of a type, a set of keys, an optional hash value, an absolute timelock and a relative timelock. The condition can be one of the symbolic values *SingleSig*, *AllSig*, *SingleSigHashLock*, *AllSigHashLock*. *SingleSig* specifies that this output can only be spent by a transaction signed using one of the keys stored in the condition. *AllSig* specifies that a spending transaction must be signed using all of the keys stored in the condition. The types with the suffix *-HashLock* specify the additional constraint that a spending transaction must provide a preimage to the hash value specified in the condition. Additionally, a spending transaction must specify timelocks with a value that is greater than or equal to the values specified in the condition. The absolute timelock enforces that a spending transaction is only valid if the transaction is published at a point in time that is greater than or equal to the value of the absolute timelock. The relative timelock enforces that a spending transaction is only valid if the spending transaction is published at a point in time at which the age of the transaction that contains the output that is spent is greater than or equal to the relative timelock. An input of a transaction consists of a reference to an output that is being spent by this input, a relative timelock, and witness data which contains of a set of signatures and an optional preimage.

Transactions in Bitcoin have ids which are calculated by hashing a transaction. For our model, we require the property that each transaction has a unique id. As explained above, we model transaction ids by choosing for each transaction an id as an integer number that has not yet been used as a transaction id. The variables *ChannelUsedTransactionIds[c]* stores a set of all transaction ids drawn to ensure that each new transaction id is unique. To reduce the number of possible states of the formalization, we specify a unique range of integer numbers for each channel c. In the constant *AvailableTransactionIds*, for each channel c this set of numbers is stored from which transaction ids created by the module *PaymentChannelUser* for channel c can be drawn.

Lightning requires users to exchange signatures on transactions. Because TLA$^+$ does not provide a native way to model signatures, we use an approach that meets the following three requirements for signatures: (1) A signature must be bound to the transaction that is signed by the signature so that a user is able to check whether a signature was created for a specific transaction or not. (2) The validity of a signature can be checked using the corresponding public key. (3) However, a signature must only be generatable using the corresponding private key. To model these requirements in a simple way, we model signing a transaction by adding the signing key to the inputs of the transaction. This makes it simple to exchange signed transactions by exchanging the whole transaction and

$$
\begin{array}{l}
FundingTransaction(fundingTxId, otherKey) \triangleq \\
\quad [id \mapsto fundingTxId, \\
\quad\ inputs \mapsto \{[parentId \mapsto Vars.FundingOutputTxId, \\
\qquad\qquad\qquad outputId \mapsto 1, \\
\qquad\qquad\qquad witness \mapsto [signatures \mapsto \{Vars.MyKey\}], \\
\qquad\qquad\qquad relTimelock \mapsto 0]\}, \\
\quad\ outputs \mapsto \{[parentId \mapsto fundingTxId, \\
\qquad\qquad\qquad\ outputId \mapsto 1, \\
\qquad\qquad\qquad\ amount \mapsto FundingAmount, \\
\qquad\qquad\qquad\ conditions \mapsto \{AllSigCondition(\{Vars.MyKey, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ otherKey\})\} \\
\qquad\qquad\quad]\}, \\
\quad\ absTimelock \mapsto 0 \\
\quad]
\end{array}
$$

Figure 3.11.: Operator to create the funding transaction. The funding transaction has a single input that references the funding output and the transaction is signed using the funder's public key. The transaction has a single output that specifies that a spending transaction has to be signed by the funder's and the other user's key.

it is trivial to verify the validity of a signature by comparing the signature to the respective public key. A caveat here is that when in Lightning only a signature is exchanged (e.g., in the `funding_signed` and `commitment_signed` messages), in the formalization the whole transaction is exchanged. In Lightning, both users must be able to create the respective transaction by themselves. To ensure in the formalization that both users have all the information necessary to create the transactions, we model the reception of a message that contains a signed transaction by comparing the received message to a record that contains the expected transaction built from locally available information. While signatures are created using private keys, we use in the following the simplifying expression 'a transaction signed by a public key k' meaning 'a transaction that has a signature that can be verified using the public key k'.

As an example, Fig. 3.11 shows a model of the funding transaction as it is used in the TLA$^+$ formalization. The id of the funding transaction is set to the parameter *fundingTxId*. The funding transaction has one input that refers to the first output of the transaction whose id is stored in *Vars.FundingInputTxId*. To spend this output, the input has as signature over the funding transaction that has been created using the key stored in *Vars.MyKey*. The input has a relative timelock of 0 which is the default value. The funding transaction has a single output. Each output contains the id of the transaction that the output belongs to. While this is redundant, it simplifies some formulas in the specification because it facilitates finding the transaction that contains a given output. Because the outputs of a transaction are stored in an unordered set, each output has an *outputId* that identifies each output and determines the order of outputs. The amount of the funding transaction's output is set to the value of *FundingBalance* which is the amount of the output that is spent by the funding transaction. The conditions of the funding transaction's output specify

```
FirstCommitTx(commitTxId, fundingTxId, myAmount) ≜
  [id ↦ commitTxId,
   inputs ↦ {[parentId ↦ fundingTxId,
              outputId ↦ 1,
              witness ↦ [signatures ↦ {Vars.MyKey}],
              relTimelock ↦ 0]},
   outputs ↦ {[parentId ↦ commitTxId,
               outputId ↦ 1,
               amount ↦ myAmount,
               conditions ↦ {SingleSigCondition(Vars.MyKey)}],
              [parentId ↦ commitTxId,
               outputId ↦ 2,
               amount ↦ DetailVars.Capacity − myAmount,
               conditions ↦ {[SingleSigCondition(DetailVars.OtherKey) EXCEPT
                                !.relTimelock = TO_SELF_DELAY],
                             AllSigCondition({Vars.MyKey,
                                              DetailVars.OtherRKey})}
              ]},
   absTimelock ↦ 0
  ]
```

Figure 3.12.: Specification of the initial commitment transaction. This transaction has a single input that references the output of the funding transaction. The commitment transaction is pre-signed by the user creating the transaction and sent to the other user who could sign and publish the transaction. The commitment transaction has two outputs: one output that is spendable by the funder and one output that can be spent by the other party after a delay or by the funder and the other party's revocation key.

that the output can be spent by a transaction that is signed using both keys *Vars.MyKey* and *otherKey*.

Similarly, Fig. 3.12 shows the formalization of the initial commitment transactions. A transaction created with the operator *FirstCommitTx* is sent to the other party so that the other party could close the channel by signing and publishing the transaction. The initial commitment transaction has a single input that references the funding transaction's output. The input is signed only by the key *Vars.MyKey*. Thus, the initial commitment transaction is not valid until it has been signed by the other party. The initial commitment transaction contains two outputs: The first output contains the funds of the party who creates the commitment transaction. Note that this is not the party who publishes the transaction because the transaction created by the operator *FirstCommitTx* is sent to the other party. Thus, the first output can be spent by a transaction that is signed by the key *Vars.MyKey*. The second output contains the funds for the other party who can publish the commitment transaction after signing it. Thus, this output needs to allow for revocation and, therefore, the output can be spent either by the other party's key stored in *DetailVars.OtherKey* after a relative timelock of *TO_SELF_DELAY* or by a revocation transaction that is signed by both keys *Vars.MyKey* and *DetailVars.OtherRKey*.

We model the blockchain as a set of transactions stored in the variable *LedgerTx*. We model publishing a transaction on the blockchain by a single step that happens instantly and includes the published transaction in the set *LedgerTx*, i.e., we assume that users have blockchain connectivity and we make the simplifying assumption that each transaction to be published is included in the next block being created. For all transactions that are added to *LedgerTx*, the domain of the function *TxAge* is expanded. Each value *TxAge[i]* is a clock that models the time since the transaction with id i was included in the blockchain. If a transaction is published, the transaction's entry in *TxAge* is initialized to 0. When time is increased by a value d, all entries in *TxAge* are increased by the value d.

3.3.2.6. Onion Routing

> In Lightning, the sender of a multi-hop payment chooses the payment's route. To improve the privacy of payments, the sender does not communicate directly with the intermediary hops on a payment's route but Lightning uses an onion routing protocol with the following idea: When adding an HTLC to the channel between the payment's sender and the first hop on the payment's route, the payment's sender adds a message that contains an identifier of the channel on which the payment should be forwarded to the second hop, the amount to forward, the timelock for the HTLC, and an encrypted message that can be decrypted by the second hop. The amount to forward is specified because typically the amount received by a hop is larger than the amount that the hop should forward. The difference is a fee that may be kept by the hop. The encrypted message for the second hop contains again an identifier for the channel on which the payment should be forwarded to the next hop, the amount to forward, the timelock for the HTLC, and an encrypted message for the next hop. These encrypted messages are nested until the innermost message is the encrypted message for the payment's receiver. [Lig26c, BOLT #4]

As onion routing is an integral part of multi-hop payments, we also formalize the onion messages used by Lightning. However, because we do not focus on privacy, we model the onion messages as unencrypted nested records. The sender of a payment creates a record that contains a layer for each hop up to the receiver of the payment. Each hop extracts one layer of this record that contains the data required to forward the payment and forwards the nested record to the next hop. Because we do not model fees, in the formalization the amount to forward is always the amount that a hop has received.

> To send a multi-hop payment, the receiver of a payment needs to transmit an invoice to the payment's sender. Such an invoice can but must not include the amount for which a payment is requested. Including the amount is optional to enable the use of invoices for use-cases such as donations in which a payment's sender independently decides on the amount to pay and the payment's receiver accepts any payment amount. Such a scenario in which the payment's receiver does not know which amount to expect requires a special security mechanism to defend against the following attack. The last hop before the payment's receiver might guess that it is the last hop (e.g., by the size

of the forwarded onion message) and add an HTLC to the channel with the payment's receiver with a much lower value than the actual payment. Because the payment's receiver does not know the expected amount, the receiver accepts the HTLC and fulfills the HTLC by sending the preimage. Thereby, the receiver receives less than the actual payment and the last hop before the receiver gains the difference. A similar attack has been described as CVE-2020-26896 [Fro20].

To protect against such an attack, the payment's receiver includes a random value called payment secret alongside the preimage in the invoice and internally stores a mapping from the preimage to the payment secret. The payment's sender includes this payment secret alongside the payment's amount in the innermost part of an onion message that is only readable by the payment's receiver. When the payment's receiver receives the onion message, the payment's receiver compares the received payment secret to the locally stored payment secret. Because the last hop before the payment's receiver cannot read the payment secret, the last hop cannot create an encrypted message that contains a wrong amount and the correct payment secret. This prevents the attack and ensures that the payment's receiver receives the amount that the payment's sender has intended to send.

In the formalization, we model the payment secret as well: The payment secret is created with the payment's preimage when an invoice is generated and the payment secret is included in an onion message and verified by the payment's receiver.

3.3.2.7. HTLC States

Each user of a payment channel needs to store information about the state of each HTLC. The state of an HTLC describes, for example, whether an HTLC is new, who has committed to the HTLC, or whether the HTLC has been fulfilled.

The official specification only sparsely defines the states of an HTLC and explicitly defines only what it means when an HTLC is irrevocably committed. An HTLC is irrevocably committed for a user A if user A has received the other user's signature on the commitment transaction containing the HTLC and has received the secrets required for revocation for all transactions that do not contain the HTLC and were signed by user A.

The changes of the state of an HTLC in the formalization are described by the state machine shown in Fig. 3.13. These states of HTLCs can be mapped to those used in Core Lightning[3], an implementation of Lightning. Figure 3.14 shows how the first states correspond to the intermediate states of an update of a payment channel when an HTLC is added (see Fig. 2.7 on page 19). When an HTLC is created, the state of the HTLC is initialized to NEW. Depending on whether the HTLC is an outgoing or an incoming HTLC, a user first receives the other user's commitment transaction that contains the HTLC or first sends

[3] `https://github.com/ElementsProject/lightning/blob/master/common/htlc_state.h`

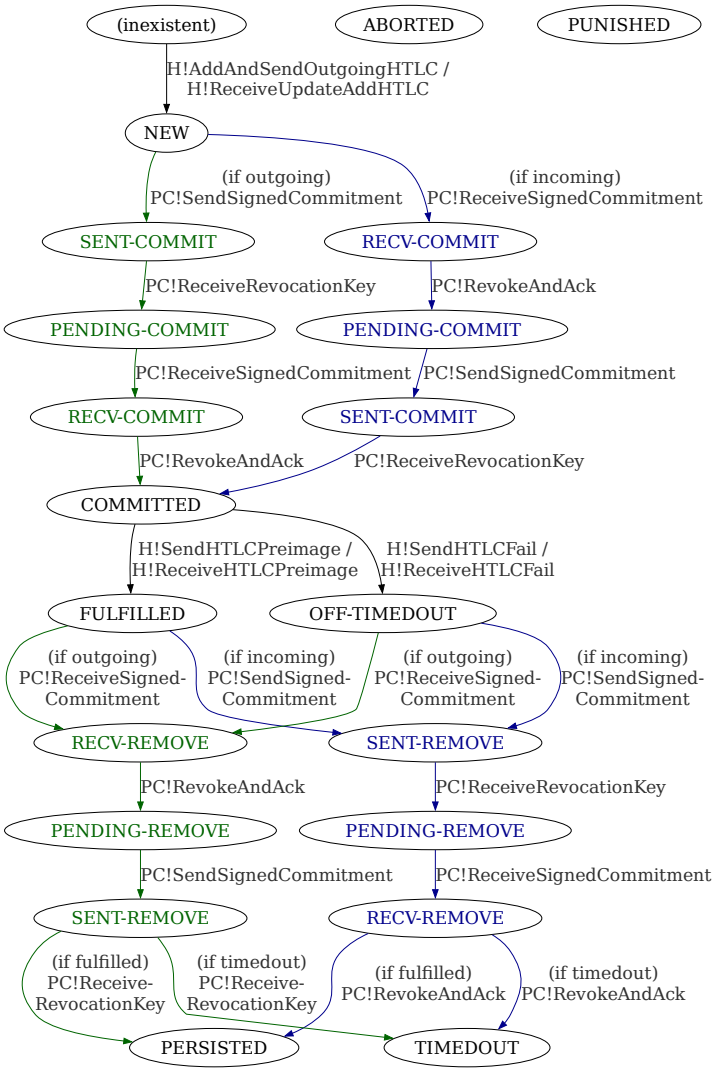

Figure 3.13.: Flow chart of HTLC states. The actions with the prefix H! are actions of the *HTLCUser* module (see Section 3.3.4); those prefixed with PC! are actions of the *PaymentChannelUser* module (see Section 3.3.3). Outgoing HTLCs follow the path printed in green; incoming HTLCs follow the path printed in blue. The flow chart shows all possible off-chain state transitions. The states ABORTED and PUNISHED can only be reached if a channel is closed. When a channel is closed, transitions are possible that skip states and transition directly to a final state (PERSISTED, TIMEDOUT, ABORTED, or PUNISHED). In the abstraction of payment channels (see Section 4.4), the states printed in blue and green are omitted and only the states printed in black are considered (see Fig. 4.25).

a commitment transaction containing the HTLC to the other user. Consequently, the order of the following states of an HTLC is dependent on whether the HTLC is incoming or outgoing. When a commitment transaction is received that contains the HTLC, the HTLC's state is advanced to RECV-COMMIT. When the revocation key for the previous commitment transaction is sent to the other party, the HTLC's state is changed to PENDING-COMMIT or COMMITTED depending on whether a commitment transaction containing the

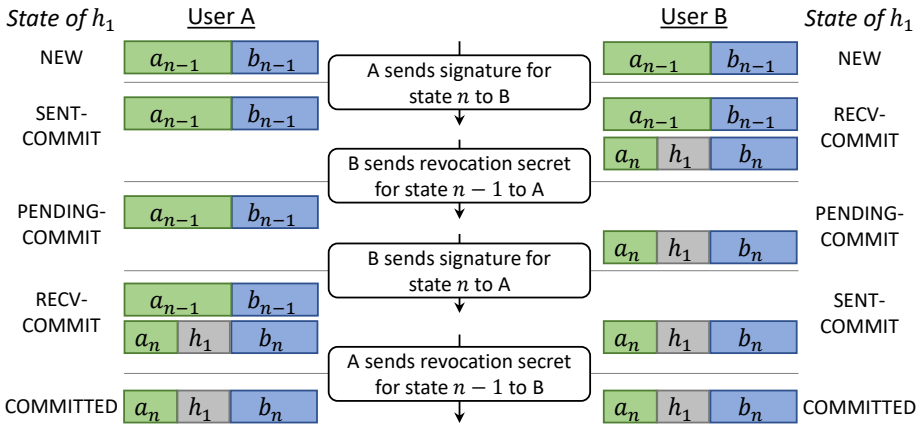

Figure 3.14.: Visualization of intermediate states of a new HTLC that is being committed. The figure shows the state of two users: User A on the left side and user B on the right side. The figure contains five lines that correspond to five different states. In the center are descriptions of the steps between each pair of consecutive states. The outermost column shows the state that the HTLC being added has in the variables of the respective user. The colored diagram visualizes the commitment transactions that a user can publish in a specific state without being punishable. For each commitment transaction, the amounts of the transaction's outputs are visualized. For example, in the first state both users have a commitment transaction that sends a_{n-1} coins to user A and b_{n-1} coins to user B. In the last state, both users have a commitment transaction that sends a_n coins to user A, b_n coins to user B, and h_1 coins are locked in the HTLC. In the intermediate states, there are states in which a user has two commitment transactions and could publish either of these transactions without being punished.

HTLC has already been sent to the other party. After the HTLC has been included in a new commitment transaction sent to the other party, the HTLC's state is advanced to SENT-COMMIT and, when the other party's revocation key for the outdated commitment transaction is received, the HTLC's state is advanced to COMMITTED or PENDING-COMMIT. When an HTLC has reached the state COMMITTED, an HTLC is irrevocably committed as specified by the official specification of Lightning, i.e. both parties have a commitment transaction that contains the HTLC and is signed by the other party and both parties have revoked all previous commitment transactions.

An HTLC that is in state COMMITTED can either be fulfilled which changes the HTLC's state to FULFILLED or, after the HTLC's timelock has passed, the HTLC can be timed out changing the HTLC's state to OFF-TIMEDOUT (for 'off-chain timed out'). The process for the removal of an HTLC is analogous to the addition of an HTLC described above and uses the intermediate HTLC states SENT-REMOVE, PENDING-REMOVE, and RECV-REMOVE. After an HTLC has been removed, the HTLC is either in the state PERSISTED if the HTLC has been fulfilled or in the state TIMEDOUT if the HTLC has been timed out.

Before an HTLC is committed, an HTLC can be aborted. In this case, the HTLC's state is changed to ABORTED. When one of the users has cheated and is punished, the state of all HTLCs is set to PUNISHED to indicate that the cheating user's total balance in the channel has been received by the punishing user. The HTLC states PERSISTED, TIMEDOUT, ABORTED, and PUNISHED are the four final states of an HTLC. Once an HTLC is in one of

these states, the HTLC's state does not change anymore. In the formalization, each HTLC is finally in one of these final states because each payment channel is closed and every potential inconsistency between the two users of a channel is resolved by the transactions on the blockchain.

3.3.3. Specification of Payment Channels

The module *PaymentChannelUser* contains over 20 actions that describe possible steps inside a payment channel. In the following, we present these actions starting with actions for opening a channel, then actions for updating a channel, and finally actions for closing a channel honestly and dishonestly. As an illustrative example of how the actions are specified, we describe the first actions in greater detail and explain the underlying principles and ideas. The remaining actions are described more briefly.

We start by giving an overview of the variables of the module *PaymentChannelUser*. The module has access to the global variables *LedgerTx*, *LedgerTime*, and *TxAge*. With these variables, users can read the state of the ledger, publish transactions on the ledger, and read the current time. An instance of the module *PaymentChannelUser* is parameterized for a specific user u and a payment channel c of user u. In the following, we use u and c to refer to this user and channel and we refer to u as the current user and to c as the current channel. The module *PaymentChannelUser* has access to the user variables of user u, to the channel variables of channel c, and to the ⟨channel and user⟩ variables of channel c and user u. Because only variables of user u and channel c can be accessed, we use shorter names for these variables. For the user variables, we omit the prefix *User* and omit the index *[u]*. Consequently, the user variables relevant for the module *PaymentChannelUser* are *ChannelBalances*, *Payments*, *ExtBalance*, *Honest*, and *PreimageInventory*. Analogously, the channel variables are named *Messages*, *UsedTransactionIds*, and *PendingBalance*, and the ⟨channel and user⟩ variables used in the module *PaymentChannelUser* are *Balance*, *State*, *Vars*, *DetailVars*, and *Inventory*.

3.3.3.1. Steps of PaymentChannelUser: Opening a Channel

> To initiate the opening protocol for a payment channel, the funder of the payment channel sends an `open_channel` message to the user to whom the channel should be opened. The `open_channel` message contains several fields for the parameterization of the channel.

The sending of the `open_channel` message is modeled by the action *SendOpenChannel* shown in Fig. 3.15. Most actions of the module *PaymentChannelUser* are enabled for a user u only if the user u is in a specific state of the protocol. This state is stored in the variable *State* and is initially set to `init`. The actions update the state to a new value; e.g., the action *SendOpenChannel* sets the state to `open-sent-open-channel`. The variable *State* indicates for a user u what messages to expect and what messages may be sent at a specific point in the protocol execution. When a user is in the state `open-sent-open-channel`,

$$
\begin{aligned}
&SendOpenChannel \triangleq \\
&\quad \land\ State = \text{“init”} \\
&\quad \land\ State' = \text{“open-sent-open-channel”} \\
&\quad \land\ FundingAmount > 0 \\
&\quad \land\ ExtBalance \geq FundingAmount \\
&\quad \land\ SendMessage([EmptyMessage \text{ EXCEPT } !.recipient = OtherUser, \\
&\qquad\qquad\qquad\qquad\qquad\qquad !.type = \text{“OpenChannel”}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad !.data.key = Vars.MyKey, \\
&\qquad\qquad\qquad\qquad\qquad\qquad !.data.capacity = FundingAmount, \\
&\qquad\qquad\qquad\qquad\qquad\qquad !.data.rKey = DetailVars.MyRKey]) \\
&\quad \land\ DetailVars' = [DetailVars \text{ EXCEPT } !.Capacity = FundingAmount]
\end{aligned}
$$

Figure 3.15.: Action to send an `open_channel` message. The action is enabled if the current user's state equals `init` and the user's external balance is positive. The user's state is advanced to `open-sent-open-channel`, the operator *SendMessage* is used to append an `open_channel` message to the *ChannelMessages* variable, and the value of the user's external balance is stored as capacity of the channel.

the user expects to receive an `accept_channel` message in the next step. Figure 3.15 also shows that an `open_channel` message is sent to the fundee proposing to open a channel with the value of the constant *FundingAmount* as capacity. The constant *FundingAmount* is an externally defined parameter of the specification (see Fig. 4.36 on page 168 where the constant is defined for all users and channels as *UserChannelFundingAmount*). While the `open_channel` message contains more fields, most of the message's fields are ignored for our formalization because they are not necessary (see Tables A.1 and A.2). The operator *SendMessage* adds the name of the sender to the given message and then adds the message to the receiver's message queue. The variable *DetailVars* is used by the module *PaymentChannelUser* to store low-level information related to the payment channel. Here, in the action *SendOpenChannel*, the channel's capacity is stored in the field *DetailVars.Capacity*.

> If the receiver wants the channel to be opened, the receiver replies with an `accept_channel` message (see Tables A.3 and A.4) in which the receiver sends their parameters for the channel to the funder. This includes the key material required to build the initial commitment transaction.

The sending of the message `accept_channel` is modeled by the action *SendAcceptChannel* (not shown in a figure). The `open_channel` and `accept_channel` messages contain many fields that are not relevant for the TLA$^+$ formalization (see Tables A.1 to A.4). One reason for fields not being modeled is that they are used for features that are not included in the TLA$^+$ formalization because they are optional features, e.g, one option allows for the funder to assign a share of the channel's funds to the fundee directly when the channel is opened. Further, the TLA$^+$ formalization does not model transaction fees for on-chain transactions. We leave this aspect out of scope to keep the model and security analysis focused. However, modeling transaction fees could be included in the formalization as part of future work.

$$
\begin{aligned}
&\mathit{CreateFundingTransaction} \triangleq \\
&\quad \land\ \mathit{State} = \text{“open-sent-open-channel”} \\
&\quad \land\ \mathit{State}' = \text{“open-funding-created”} \\
&\quad \land\ \exists\, \mathit{message} \in \mathit{MyMessages} : \\
&\qquad \land\ \mathit{message.recipient} = \mathit{MyName} \\
&\qquad \land\ \mathit{message.type} = \text{“AcceptChannel”} \\
&\qquad \land\ \mathit{RemoveReceivedMessage}(\mathit{message}) \\
&\qquad \land\ \textsc{let}\ \mathit{fundingTxId} \triangleq \mathit{TIO!GetNewTransactionId} \\
&\qquad\qquad\ \ \mathit{fundingTransaction} \triangleq \mathit{FundingTransaction}(\mathit{fundingTxId},\ \mathit{message.data.key}) \\
&\qquad\ \textsc{in}\ \ \land\ \mathit{TIO!MarkAsUsed}(\mathit{fundingTxId}) \\
&\qquad\qquad\ \land\ \mathit{Inventory}' = [\mathit{Inventory}\ \textsc{except} \\
&\qquad\qquad\qquad\ !.\mathit{transactions} = @ \cup \{\mathit{fundingTransaction}\}] \\
&\qquad\qquad\ \land\ \mathit{DetailVars}' = [\mathit{DetailVars}\ \textsc{except} \\
&\qquad\qquad\qquad\ !.\mathit{OtherKey} = \mathit{message.data.key}, \\
&\qquad\qquad\qquad\ !.\mathit{OtherRKey} = \mathit{message.data.rKey}, \\
&\qquad\qquad\qquad\ !.\mathit{fundingTxId} = \mathit{fundingTransaction.id}]
\end{aligned}
$$

Figure 3.16.: The action is enabled if the current user's state equals `open-sent-open-channel` and there exists an incoming `accept_channel` message for the current user. A funding transaction is created using the operator *FundingTransaction* (see Fig. 3.11 on page 62), the funding transaction's id is stored as being used, and the funding transaction is stored in the current user's *Inventory*.

> With the information received with the `accept_channel` message, the funder creates the funding transaction and the first commitment transaction. The funder sends the `funding_created` message (see Table A.5) that contains the funder's signature of the first commitment transaction and the id of the funding transaction which is required for creating commitment transactions.

The reception of the `accept_channel` message and the creation of the funding transaction is modeled by the action *CreateFundingTransaction* shown in Fig. 3.16. The action uses the operator *TIO!GetNewTransactionId* to create a new transaction id that is unique from previously created transaction ids and creates the funding transaction using the operator *FundingTransaction* shown in Fig. 3.11 on page 62. Using the operator *TIO!MarkAsUsed(fundingTxId)*, the transaction id is marked as being used by storing the transaction id in the set *UsedTransactionIds*. Finally, the created transaction is stored in the user's inventory. Creating and sending the initial commitment transaction is modeled by the action *SendSignedFirstCommitTransaction* (not shown in a figure).

> The fundee receives the `funding_created` message, validates and stores the signature. The fundee creates the version of the first commitment transaction that the funder can publish and sends a signature for the first commitment transaction to the funder in the `funding_signed` message (see Table A.6).

The reception of the `funding_created` message and the creation and sending of the first commitment transaction for the funder is modeled by the action *ReplyWithFirstCommitTransaction* (not shown in a figure).

$$
\begin{aligned}
&PublishFundingTransaction \triangleq \\
&\quad \land\ State = \text{“open-recv-commit”} \\
&\quad \land\ State' = \text{“open-funding-pub”} \\
&\quad \land\ \exists\, tx \in Inventory.transactions : \\
&\qquad tx.id = DetailVars.fundingTxId \\
&\quad \land\ \textsc{let}\ fundingTx \triangleq \textsc{choose}\ tx \in Inventory.transactions : \\
&\qquad\qquad\qquad\qquad tx.id = DetailVars.fundingTxId \\
&\qquad fundingTxSigned \triangleq SignTransaction(fundingTx,\ Vars.MyKey) \\
&\quad \textsc{in}\quad \land\ Ledger!PublishTx(fundingTxSigned) \\
&\qquad\quad \land\ Inventory' = [Inventory\ \textsc{except} \\
&\qquad\qquad !.transactions = @ \setminus \{fundingTx\}] \\
&\quad \land\ ExtBalance \geq FundingAmount \\
&\quad \land\ ExtBalance' = ExtBalance - FundingAmount \\
&\quad \land\ Balance' = Balance + FundingAmount
\end{aligned}
$$

Figure 3.17.: If the current user's state equals `open-recv-commit`, the funding transaction exists in the user's *Inventory*, and the user's external balance is positive, the funding transaction is signed using the operator *SignTransaction* and published on the blockchain using the operator *Ledger!PublishTx*. The funding transaction is removed from the user's *Inventory* and the balances are updated to reflect that the channel has been opened and the external balance has moved into the payment channel.

$$
\begin{aligned}
&NoteThatFundingTransactionPublished \triangleq \\
&\quad \land\ State = \text{“open-sent-commit”} \\
&\quad \land\ State' = \text{“open-funding-pub”} \\
&\quad \land\ \exists\, tx \in LedgerTx : \\
&\qquad \land\ tx.id = DetailVars.fundingTxId \\
&\qquad \text{Verify that the funding transaction uses the correct capacity:} \\
&\qquad \land\ \exists\, output \in tx.outputs : \\
&\qquad\quad \land\ \exists\, condition \in output.conditions : \\
&\qquad\qquad condition.data.keys = \{Vars.MyKey,\ DetailVars.OtherKey\} \\
&\qquad\quad \land\ output.amount = DetailVars.Capacity \\
&\quad \land\ Balance' = 0
\end{aligned}
$$

Figure 3.18.: The action *NoteThatFundingTransactionPublished* is enabled for the fundee, who does not publish the funding transaction, if the fundee is in the state `open-sent-commit` and there exists a transaction in the ledger that has the funding transaction's id and this transaction has an output with an amount that matches the expected capacity of the payment channel.

> Having verified and stored the signature of the first commitment transaction, the funder publishes the funding transaction. After both users have noted that the funding transaction was published and confirmed on the blockchain, both users send each other a `channel_ready` message which indicates that the channel can now be used to process payments. The `channel_ready` message (see Table A.7) contains the key material that is required to derive the public keys required to create the second commitment transaction.

The reception and verification of the first commitment transaction is modeled by the action *ReceiveCommitTransaction*. The action *PublishFundingTransaction* (see Fig. 3.17) models the publication of the funding transaction. To publish the transaction, the operator

$$
\begin{array}{l}
SendNewRevocationKey \triangleq \\
\quad \land \lor State = \text{“open-funding-pub”} \land State' = \text{“open-new-key-sent”} \\
\qquad \lor State = \text{“open-new-key-received”} \land State' = \text{“idle”} \\
\quad \land \textsc{let}\ newRevocationKey \triangleq [DetailVars.MyRKey\ \textsc{except}\ !.Index = @ + 1] \\
\qquad \textsc{in}\quad \land SendMessage([EmptyMessage\ \textsc{except} \\
\qquad\qquad\qquad\qquad\qquad !.recipient = NameForUserID[OtherUserID], \\
\qquad\qquad\qquad\qquad\qquad !.type = \text{“ChannelReady”}, \\
\qquad\qquad\qquad\qquad\qquad !.data.rKey = newRevocationKey]) \\
\qquad\qquad \land DetailVars' = [DetailVars\ \textsc{except} \\
\qquad\qquad\qquad\quad !.MyNewRKey = newRevocationKey] \\
\qquad\qquad \land Inventory' = [Inventory\ \textsc{except} \\
\qquad\qquad\qquad\quad !.keys = @ \cup \{newRevocationKey\}]
\end{array}
$$

Figure 3.19.: With the action *SendNewRevocationKey*, a new revocation key is created by increasing the *Index* value of the previous revocation key. The new revocation private key is stored in the *Inventory* and the public revocation key is sent to the other user in the `channel_ready` message.

Ledger!PublishTx is used that adds the given transaction to the variable *LegerTx* and updates the variable *TxAge* accordingly. Because the publication of the funding transaction actually opens the payment channel by spending the funding output that is owned by the funder, the funder's balances are adjusted: The external balance is reduced to by the constant *FundingAmount* and the variable *Balance* which is the user's balance in the current channel is increased by the capacity of the newly opened channel. Once the transaction is included in *LedgerTx*, the action *NoteThatFundingTransactionPublished* (see Fig. 3.18) is enabled for the fundee. With this action, the fundee learns that the channel has been opened and initializes the user's balance in the channel to 0. To ensure that the funding transaction matches the capacity that was announced in the `open_channel` message, the funder compares the amount of the funding transaction's output to the stored amount for the channel's capacity.

After the funder has published the funding transaction, the action *SendNewRevocationKey* (see Fig. 3.19) is enabled for the funder and after the fundee has noted that the funding transaction was published, the action *SendNewRevocationKey* is enabled for the fundee. With this action, both users send each other a `channel_ready` message that includes the public revocation key required to build the second commitment transaction. The creation of the new public revocation key is modeled by incrementing the index of the previous public revocation key by 1. The reception of the `channel_ready` message is modeled by the action *ReceiveNewRevocationKey*. Because each user can send the new revocation public key to the other user after noticing that the funding transaction has been published, the order in which the `channel_ready` messages are sent is not deterministic and a user might first send the `channel_ready` message and then receive the other user's `channel_ready` message or the other way around. In the formalization, this has the effect that the actions *SendNewRevocationKey* and *ReceiveNewRevocationKey* are enabled in two different states (e.g., `open-funding-pub` and `open-new-key-received`) and update the state accordingly (e.g., `open-new-key-sent` and `idle`). After the new revocation public keys are exchanged, the channel is ready to operate, i.e., ready to add HTLCs to the commitment transaction.

3.3.3.2. Steps of PaymentChannelUser: Channel Update

> After a user has sent at least one `update_add_htlc` message (see Table A.8) to inform the other user about an HTLC to be added, the user sends a `commitment_signed` message. The `commitment_signed` message contains a signature for the new commitment transaction. In the new commitment transaction, all outgoing HTLCs for which an `update_add_htlc` message was sent are included and incoming HTLCs that have been fulfilled are not included. For all HTLCs included in the commitment transaction, the `commitment_signed` message contains also a signature for the HTLC success or timeout transaction.

When the user u is in state `idle`, the action *SendSignedCommitment* (not shown in a figure) can be enabled if there is at least one HTLC to add or remove and the action *ReceiveSignedCommitment* can be enabled if there is a `commitment_signed` message in the variable *Messages* to be received by user u. The action *SendSignedCommitment* creates a new commitment transaction based on the current commitment transaction. The new commitment transaction includes all HTLCs that were included in the current commitment transaction and have not yet been fulfilled and all HTLCs that are in state NEW and have not timed out. The action *SendSignedCommitment* models the sending of the signatures of the new commitment transaction and the HTLC transactions by sending the signed transactions in a `commitment_signed` message to the other user. The state of the HTLCs stored in *Vars.IncomingHTLCs* and *Vars.OutgoingHTLCs* is updated according to Fig. 3.13 and the user's *Balance* is reduced by the amount of every outgoing HTLC that is added to the commitment transaction. The amount that is deducted from the user's *Balance* is added to the channel's *PendingBalance*.

The action *ReceiveSignedCommitment* models the reception of the `commitment_signed` message. While in Lightning the received signature is validated, in the formalization the received transaction is validated by comparing it to a local computation of the expected commitment transaction. This check does not only protect against dishonest behavior but also a commitment transaction that commits to an incoming HTLC that has timed out will not be accepted by the action *ReceiveSignedCommitment*. The state of the HTLCs stored in *Vars.IncomingHTLCs* and *Vars.OutgoingHTLCs* is updated according to Fig. 3.13.

> The receiving user responds with a `revoke_and_ack` message (see Table A.9) to acknowledge the reception of the new commitment transaction signature and to revoke the old commitment transaction by sending the secret required to create the private revocation key for the old commitment transaction.

Sending a `revoke_and_ack` message is modeled by the action *RevokeAndAck*. This action sends the private revocation key for the old commitment transaction to the other user and updates the states of HTLCs according to Fig. 3.13 on page 66. The action *ReceiveRevocationKey* models the reception of a `revoke_and_ack` message by storing the private revocation key and updating states of HTLCs according to Fig. 3.13. The action *ReceiveRevocationKey* adds the amount of HTLCs that failed and were removed to the user's *Balance* and removes the amount from the channel's *PendingBalance*.

3.3.3.3. Steps of PaymentChannelUser: Channel Closing

We have discussed how a channel is opened and updated. While in Lightning a channel can be closed either by one party publishing a commitment transaction or by creating a dedicated closing transaction, we model only the first way of closing a channel. The creation of a dedicated closing transaction is an optimization to avoid delays and reduce the amount of fees paid but the optimization is not required for a secure payment channel. Thus, to simplify the TLA$^+$ specification of Lightning, we model that a channel is always closed by publishing a commitment transaction which is a way that inherently must be possible. In contrast to using a dedicated closing transaction, when a channel is closed with a commitment transaction the other party does not expect the channel to be closed and a channel update might currently be in progress. This makes specifying the closing of a channel a challenging process because many edge cases need to be handled. A further challenge for writing the TLA$^+$ formalization of Lightning is that the official specification of Lightning does not specify the closing of a channel as clear as opening and updating a channel. The reason for this is that while opening and updating require peer-to-peer communication, closing a channel requires only communication between a user and the blockchain and, because the official specification of Lightning focuses on the peer-to-peer communication, the specification leaves the details up to the implementations of Lightning. While the official specification of Lightning gives recommendations how to handle on-chain transaction [Lig26c, BOLT #5], every implementation of Lightning has to find a concrete way how to implement these recommendations.

We model that the latest commitment transaction is published on the blockchain with the action *CloseChannel*. The action signs the latest commitment transaction, adds the transaction to *LedgerTx*, and changes the user's state to `closing`. If one user has published the latest commitment transaction, the action *NoteThatOtherUserClosedHonestly* is enabled for the other user. With the action *NoteThatOtherUserClosedHonestly*, the user's state changes to `closing`. Both actions, *CloseChannel* and *NoteThatOtherUserClosedHonestly*, also adjust the user's *Balance* and the state of HTLCs in the following way: The value of a user's *Balance* is increased by the amount of those HTLCs that were already in the process of being added but are not included in the published commitment transaction. The states of HTLCs is adjusted depending on whether they are included in the published commitment transaction: If an HTLC is included in the transaction, the HTLC's state is set to COMMITTED. If an HTLC is not included, there are three possible outcomes: If the HTLC has not been committed, the state of the HTLC is set to ABORTED. If the HTLC was committed but not fulfilled, the HTLC's state is set to TIMEDOUT. And if the HTLC was fulfilled, the HTLC's state is set to PERSISTED.

When a published commitment transaction includes HTLCs, these HTLCs need to be resolved on-chain by spending the outputs of the HTLCs either with the corresponding preimage or without the preimage after an HTLC has timed out. The action *ResolveHTLCAfterClose* models the publication of either the publication of HTLC success and HTLC timeout transactions that are stored in a user's *Inventory* or of transactions that can be created with the keys known by a user. The action adjusts the state of the corresponding

HTLCs. If the action *ResolveHTLCAfterClose* has published the preimage of an HTLC that was not previously off-chain fulfilled, the user's *Balance* and the channel's entry in the user's *ChannelBalances* are increased by the amount of the HTLC. If the user was the final receiver of the payment, the payment's state in *Payments* is updated to `processed`.

After one user has published a transaction that spends an HTLC output of the published commitment transaction, the other user can observe this and update their state accordingly. To handle these observations, there are the actions *NoteThatHTLCTimedOutOnChain* and *NoteThatHTLCFulfilledOnChain*. The action *NoteThatHTLCTimedOutOnChain* is enabled if an HTLC has been timed out on-chain and the local state of the HTLC does not yet reflect this situation. The state of such an HTLC is set to TIMEDOUT and, if necessary, the variables *Balance* and *PendingBalance* are updated. The action *NoteThatHTLCFulfilledOnChain* is enabled if the other user has fulfilled an HTLC on-chain. The action retrieves the preimage from the transaction that spends the HTLC output and stores the preimage in *PreimageInventory*. The action *NoteThatHTLCFulfilledOnChain* also updates the variables *Balance* and *PendingBalance* if necessary and, if an HTLC was fulfilled that belonged to a payment that was sent, the variables *Payments* and *ChannelBalances* are updated to reflect that the payment has been processed.

A further way to close a channel is to cheat. We model cheating with the action *Cheat*. The action is enabled only if *Honest* is FALSE. There are two different types of transactions that can be published by the action *Cheat*: Either an outdated commitment transaction is published from the current user's *Inventory* or a new transaction is created that uses the current user's keys and preimages to create a transaction that spends an output of a transaction that was published on the blockchain. Publishing an outdated commitment transaction is an obvious way to cheat and is straightforward to formalize. Spending outputs by creating new transactions is more difficult to formalize in a way that can be efficiently model checked but this type of cheating is important to find flaws in the protocol that allow a user to spend an output that is not intended to be spent by the user. We model this cheating in the following way: The first step is to find all transactions in *LedgerTx* that do not belong to another channel of the current user. The transactions in another channel are considered by the action *Cheat* of the respective channel. For all outputs in the remaining transactions, it is checked whether a valid input can be created using the keys and preimages known to the current user. All valid inputs that have been found are used as inputs for a transaction that has an output that is spendable only by the current user. If this transaction spends an output that is not already exclusively spendable by the current user then the transaction is published by the action *Cheat*.

If one user has published an outdated commitment transaction, the other user might punish the user by publishing a revocation transaction. This is specified by the action *Punish*. The action *Punish* checks if *LedgerTx* contains an outdated commitment transaction and possibly HTLC timeout and success transactions that spend an HTLC output of the outdated commitment transaction. If such outdated transactions are found, the action creates a revocation transaction that spends all outputs of the outdated commitment transaction and possibly of the HTLC transactions using the keys known by the punishing

user. As the user has the private revocation keys required to spend those transaction outputs, such a revocation transaction can be created and is published on the blockchain.

To update the state of HTLCs after a cheating transaction has been published, the action *NoteAbortedHTLCs* sets the state of all HTLCs that are in state NEW to state ABORTED. If a user publishes their latest commitment transaction, this step is not required because the states of HTLCs are already updated in the actions *CloseChannel* and *NoteThatOtherUserClosedHonestly*.

To ensure that a user's local state of all HTLCs matches the HTLC states on the blockchain, the user's state transitions to `closed` only if all HTLCs have reached a final HTLC state, i.e. ABORTED, TIMEDOUT, or PERSISTED. To set a user's state to `closed`, the module *PaymentChannelUser* contains the action *Terminate*. This action is enabled if the user's state is `closing`, the user has no pending messages to receive, and all HTLCs are in a final state. If the current user has cheated or punished, the state of all HTLCs is set to PUNISHED and the user's *Balance* is set to either 0 if the user has cheated or to the channel's capacity if the user has punished. The action *Terminate* updates the state of HTLCs and the *Balance* assuming that the punishment was successful without further verification. We chose this approach to simplify the abstraction of payment channels. With this approach, it suffices in the abstraction to change the state of the HTLCs to PUNISHED if a payment channel is closed by cheating. We verify that the punishment is actually successful by verifying the security property. For the verification of the security property, the actual balances on the blockchain are compared to the initial balances and the state of payments and it is verified that an honest user has at least their correct balance. The final state of HTLCs is not further relevant.

3.3.4. Specification of Multi-Hop Payments

The module *HTLCUser* models the steps required to perform a multi-hop payment. There are three actions of the module *HTLCUser* that are specific to a user but not to a specific channel and the other actions of the module are specific to a user and a channel. The first actions are used to request, generate, and receive an invoice for a payment. The other actions are used to to add an HTLC, send the preimage, or fail an HTLC.

The module *HTLCUser* uses the global variables *LedgerTime* and *Messages* and several variables that are specific to the current user or the current channel. As above for the module *PaymentChannelUser*, we use shorter identifiers for the variables that are internally used in the module *HTLCUser*. E.g., we use *Payments* for the variable *UserPayments[u]* for user u modeled by the module *HTLCUser*. The module *HTLCUser* uses the user variables *ChannelBalances*, *Payments*, *NewPayments*, *PreimageInventory*, *PaymentSecretForPreimage*, *Honest* and the channel variables *PendingBalance*, *Balance*, and *ChannelMessages*. We do not omit the prefix of the variable *ChannelMessages* to prevent confusion with the global variable *Messages*. Further, the module *HTLCUser* uses the ⟨channel and user⟩ variables *State* and *Vars*.

> In Lighting, if a user wants to send a payment, the user needs an invoice from the receiver of the payment. To create an invoice, the receiver generates two random values, a preimage and a payment secret. The receiver of the payment includes the hash of the preimage and the payment secret in the invoice and sends the invoice to the user who wants to send the payment. Further, the invoice contains additional data (see Table A.10) to describe the purpose of the payment for example.

In the formalization, we model that the sender of a payment requests an invoice for a specific payment by sending a message of type `RequestInvoice` to the payment's receiver. This step is modeled by the action *RequestInvoice* that sends the message and updates the respective payment record in *NewPayments* to record that for this payment an invoice has been requested. To send this message, the message is added to the set in the global variable *Messages*. Because the payment's sender and receiver might not share a common payment channel, there might not be a variable *ChannelMessages* over which the message could be sent. Therefore, the request for the invoice as well as the invoice itself are exchanged using the global variable *Messages*. Also in practice, the invoice might be transferred over a different communication channel, e.g. using a QR-code that is shown to the payment's sender.

The reception of the `RequestInvoice` message by the payment's receiver is modeled by the action *GenerateAndSendPaymentHash*. While in Lightning, the preimage and the payment secret are randomly drawn, in the formalization, these values are deterministically derived from the payment's id. The preimage is stored in the user's *PreimageInventory* and the user's mapping *PaymentSecretForPreimage* is extended to map the preimage to the payment secret. The hash value of the preimage and the payment secret are sent to the payment's sender in a message of type `Invoice`. The model of an invoice used in the TLA$^+$ formalization contains only the hash value, the payment secret, and a payment id. The additional data in the invoice is not included because it is not required for the security of the protocol.

An invoice request can also be ignored by the action *IgnoreInvoiceRequest* that handles the `RequestInvoice` message without any further effects.

> After the payment's sender has received the invoice, the sender extracts the payment hash and the payment secret. The payment's sender chooses a route through the Lightning Network, i.e. a list of users that are connected through payment channels and whose last hop is the payment's receiver.

The reception of the invoice is modeled by the action *ReceivePaymentHash*. This action updates the record in *NewPayments* for the payment for which the invoice was received and adds the hash value and the payment secret. In the formalization, we leave route finding out of scope and assume that the sender receives the route as external input. The action also models the creation of the onion package that contains for each hop on the route the information which hop the next hop on the payment's route is and the encrypted data that should be forwarded to the next hop. As we do not focus on privacy, we model the onion packages as unencrypted nested records of which each hop extracts one layer.

> To actually send the payment, the payment's sender sends an `update_add_htlc` message (see Table A.11) to the first hop on the path. The `update_add_htlc` message contains an id for the HTLC and the HTLC's amount, hash, and timelock. Further, the message contains an 'onion_routing_packet' that specifies the next hop and an encrypted payload for that hop which in turn contains the information about the subsequent hop, and so on.

The action *AddAndSendOutgoingHTLC* models that an HTLC for the payment is added to the channel and an `update_add_htlc` message is sent to the other user. The action chooses a payment in the set *NewPayments* for which a hash value has already been received, whose timelock has not already passed, and for which the next hop is the other user in the current channel. The action *AddAndSendOutgoingHTLC* is not only used by the sender of a multi-hop payment to add an HTLC but also by an intermediary hop that forwards a payment. To this end, the set *NewPayments* can not only contain payments that are sent by the current user but also payments that should be forwarded by the current user. Such payments that should be forwarded are added to the set *NewPayments* by the action *ForwardPayment* that is described below. The action *AddAndSendOutgoingHTLC* is only enabled if the current user's *Balance* is greater than or equal to the amount of the HTLC to be added. The action adds a new record for the added HTLC to *Vars.OutgoingHTLCs* and sends the `update_add_htlc` message to the other user in the channel.

> The next hop receives the `update_add_htlc` message and verifies that the amount and timelock are in expected ranges and that the 'onion_routing_packet' can be decrypted. If the hop receiving the `update_add_htlc` message is not the payment's receiver but an intermediate hop, the intermediate hop sends an `update_add_htlc` message to the next hop on the route after the incoming HTLC for the payment is irrevocably committed (see Section 3.3.2.7). The forwarding of the `update_add_htlc` message is repeated until the payment's receiver receives the `update_add_htlc` message.

The reception of the `update_add_htlc` message is modeled by the action *ReceiveUpdateAddHTLC*. The received HTLC is added to *Vars.IncomingHTLCs* with the initial state NEW. If the current user is the receiver of the payment associated with the received HTLC, the payment secret and the payment's amount received in the `update_add_htlc` message are compared to the stored values. If the current user is not receiver of the payment, a new record for the forwarded payment is added to the set *Vars.PaymentsWaitingForForward*.

If an HTLC for a payment in *Vars.PaymentsWaitingForForward* has reached the state COMMITTED because the HTLC's state was advanced by actions of the module *PaymentChannelUser*, the action *ForwardPayment* is enabled. The action *ForwardPayment* removes the payment from the set *Vars.PaymentsWaitingForForward* and adds the payment to the set *NewPayments*. Thereby, the payment can be forwarded by a step of the action *AddAndSendOutgoingHTLC* in another channel of the current user.

> If a user's incoming HTLC has been committed and the user knows the corresponding preimage, the user sends the preimage in an `update_fulfill_htlc` message (see Table A.12) to the other user.

Fulfilling an HTLC is modeled by the action *SendHTLCPreimage*. The action is enabled if there exists an incoming HTLC in state COMMITTED and the preimage to the HTLC is contained in *PreimageInventory* and has not already been sent to the other user. Also, the action *SendHTLCPreimage* is only enabled for an HTLC that has not already timed out. In the TLA$^+$ formalization, the `update_fulfill_htlc` message contains only the preimage because the associated HTLC can be found by hashing the preimage and looking up the HTLC by its hash value. The action *SendHTLCPreimage* changes the state of the HTLC being fulfilled to FULFILLED. By fulfilling an HTLC, the ownership of the funds associated with the HTLC changes. Therefore, the action *SendHTLCPreimage* updates the user's *Balance* and the channel's *PendingBalance*. If the HTLC being fulfilled belongs to a payment for which the current user is the receiver, the payment's state in *Payments* is updated to `fulfilled` and the channel's entry in the user's *ChannelBalances* is decreased by the payment's amount.

> If the hop receiving the `update_fulfill_htlc` message is not the payment's sender, the hop forwards the payment's preimage in an `update_fulfill_htlc` message to the previous hop. The forwarding of the `update_fulfill_htlc` message is repeated until the payment's sender receives the `update_fulfill_htlc` message.

The action *ReceiveHTLCPreimage* models the reception of the `update_fulfill_htlc` message. The receiving user stores the received preimage in the user's *PreimageInventory* and updates the HTLC associated with the received preimage to the state FULFILLED. If the current user is the sender of a payment that is associated with the fulfilled HTLC, the payment's state is updated to the state `processed` in *Payments*.

> If an incoming HTLC that is committed times out, a user sends an `update_fail_htlc` message (see Table A.13) to the previous hop. If the hop receiving the `update_fail_htlc` message is not the payment's sender: When the removal of the hop's outgoing HTLC is irrecoverably committed or the appropriate HTLC timeout transaction is confirmed on-chain, the hop sends an `update_fail_htlc` message to the previous hop. The forwarding of the `update_fail_htlc` message is repeated until the payment's sender receives the `update_fail_htlc` message. If the hop receiving the `update_fail_htlc` message is the payment's sender: When the removal of the hop's outgoing HTLC is irrecoverably committed or the appropriate HTLC timeout transaction is confirmed on-chain, the payment is finally cancelled.

An HTLC is failed in the TLA$^+$ formalization by the action *SendHTLCFail*. The action is only enabled for an incoming HTLC if the preimage is not known for the HTLC and if either the HTLC has timed out or an associated outgoing HTLC has already been failed. The state of the HTLC that is failed is set to OFF-TIMEDOUT. While the `update_fail_htlc` message contains fields for the reason of failure in Lightning, the message is modeled in

the TLA$^+$ formalization with only the id of the failed HTLC because learning the reason for failure is not relevant for the protocol's functionality or security.

3.3.5. Further Modeling Aspects

Having presented the actions describing how a channel is managed and how payments are processed, we now present how time flow is modeled, how adversarial behavior is modeled, and how liveness is ensured.

3.3.5.1. Time Flow: AdvanceLedgerTime

HTLCs as well as commitment transactions use absolute timelocks to enforce that certain actions cannot be done before a given point in time. These timelocks are specified by numbers that refer to a specific height of the Bitcoin blockchain. Because the height of the blockchain represents a logical time, we refer to the height of the blockchain as ledger time or simply as time. We model the time, i.e., height of the blockchain, with the variable *LedgerTime* as an integer number that can increase in integer steps. A step that increases *LedgerTime* is modeled as an action of the *LedgerTime* module. The specification is an example of an explicit real-time specification. A behavior of *SpecificationI* is a sequence of steps of the users and of steps that advance the time, i.e., increase the value of *LedgerTime*. Thus, the position of the steps of the module *LedgerTime* within a behavior models whether the protocol is executed slowly (steps of the module *LedgerTime* that advance time by large intervals between user steps) or quickly (many steps of users between steps of the module *LedgerTime*).

In some situations, the protocol requires a user to take an action before the blockchain has reached a certain height. In the literature, such actions are referred to as urgent actions (see [BST98]). An example of such a situation is that, if a user knows the preimage for an HTLC, the user must fulfill the HTLC before the HTLC times out. Consider the following scenario: There is an HTLC from user A to user B with a timelock at time 10. User B knows the preimage and the current time is 8. While the action for fulfilling the HTLC is enabled, the action for increasing the variable *LedgerTime* is also enabled. It is acceptable that the value of the variable *LedgerTime* is increased to 9. However, if the value of the variable *LedgerTime* was increased to 10 before user B fulfills the HTLC, user B would not have followed the protocol which requires user B to fulfill the HTLC before the HTLC's timelock. Therefore, to model honest behavior of user B, we need to model that user B performs an action before the variable *LedgerTime* reaches the value 10. If the value of the variable *LedgerTime* is 9, the action that increases the value of *LedgerTime* should not be enabled until user B has fulfilled the HTLC. To model this urgency requirement, each module specifies a set of points in time called *TimeBounds* that are the heights of the blockchain at which the user needs to perform an action at the latest. The time (resp. height of the blockchain) will not advance further than the minimal height specified by all *TimeBounds*. For the previous example, the value of *TimeBounds* of the module *PaymentChannelUser*

$$
\begin{aligned}
&AdvanceLedgerTimeI \triangleq \\
&\quad \land\ LedgerTimeInstance!AdvanceLedgerTime \\
&\quad \land\ TxAge' = [id \in \{tx.id : tx \in LedgerTx\} \mapsto \\
&\qquad\qquad TxAge[id] + LedgerTime' - LedgerTime] \\
&\quad \land\ \forall\, c \in ChannelIDs : \forall\, u \in UsersOfChannelSet(c) : \\
&\qquad \land\ CheckTxTimeBounds(TxAge',\ PaymentChannelUser(c,\ u)!TxTimeBounds)
\end{aligned}
$$

Figure 3.20.: The action *AdvanceLedgerTimeI* of *SpecificationI* wraps the action *AdvanceLedgerTime* of the module *LedgerTime* (see Fig. 3.21) and updates the variables in *TxAge* so these clocks advance by the same interval as the variable *LedgerTime*. The action is only enabled if the new values of *TxAge* meet the conditions of the *TimeBounds* specified by in the module *PaymentChannelUser*.

$$
\begin{aligned}
&AdvanceLedgerTime \triangleq \\
&\quad \land\ LedgerTime < MAX\_TIME \\
&\quad \land\ LedgerTime' \in \{i \in PossibleTimePoints \cup \{MAX\_TIME\} : \\
&\qquad\quad \land\ i > LedgerTime \\
&\qquad\quad \land\ \forall\, bound \in TimeBounds : bound \geq LedgerTime \implies i \leq bound \\
&\qquad\quad \}
\end{aligned}
$$

Figure 3.21.: The action *AdvanceLedgerTime* of the module *LedgerTime* is enabled as long as the current value of *LedgerTime* is below a constant *MAX_TIME*. The new value of time is chosen by choosing any value in the set *PossibleTimePoints* or the value *MAX_TIME* as long as the value is larger than the current value of *LedgerTime* and is lower than or equal to all time bounds that are larger than or equal to the current value of *LedgerTime*. Time bounds that are lower than *LedgerTime* are not relevant. In *SpecificationI*, the set *PossibleTimePoints* is defined as the set of all natural numbers between 1 and the constant *MAX_TIME*. In *SpecificationI*^{time}, a variant of *SpecificationI* that we will present in Chapter 4, the operator *PossibleTimePoints* will be defined by the set $relZTP$ (see Section 4.1).

would include the HTLC's timelock - 1 for each incoming HTLC that the user has the preimage for and that has not been fulfilled and persisted. After a step has been taken that removes the condition for a time bound, the height of the blockchain can advance further. For the previous example, the time bound would be removed after a series of steps in which the user fulfills the HTLC by sending the HTLC's preimage to the other user and updates the payment channel to persist the HTLC. We check that no execution of the protocol is stuck because a state is reached in which the value of *LedgerTime* equals a time bound but no action is enabled that would lead to a removal of the time bound. This check is implemented by a liveness property that verifies that each channel with at least one honest user is finally closed (see Section 3.3.5.4).

The action *AdvanceLedgerTimeI* of module *SpecificationI* is defined as shown in Fig. 3.20. In the first conjunct, it uses the action *AdvanceLedgerTime* of the module *LedgerTime* which is defined in Fig. 3.21. This action *AdvanceLedgerTime* is enabled only if the value of the variable *LedgerTime* is below the maximum time. The action specifies that the new value of the variable *LedgerTime* must be a value in the set of possible time points and that the new value of *LedgerTime* must not be greater than any element of *TimeBounds* if the current value of *LedgerTime* is not already greater than the value. The set of possible time points is defined in *SpecificationI* as the set of all natural numbers lower than a constant *MAX_TIME*. The set *TimeBounds* contains points in time until an urgent step must be taken. This

$$
\begin{aligned}
&\mathit{WithdrawBalancesAfterChannelClosed} \triangleq \\
&\quad \land\ \mathit{LedgerTime} < \mathit{TERMINATION} \\
&\quad \land\ \mathit{LedgerTime}' = \mathit{TERMINATION} \\
&\quad \land\ \forall\, c \in \mathit{ChannelIDs} : \forall\, u \in \mathit{UsersOfChannelSet}(c) : \\
&\qquad\quad \mathit{ChannelUserState}[c][u] = \text{“closed”} \\
&\quad \land\ \mathit{ExtBalance}' = [u \in \textsc{domain}\ \mathit{ExtBalance} \mapsto \mathit{ExtBalance}[u] + \\
&\qquad\qquad\qquad \textsc{if}\ \mathit{UserHonest}[u] \\
&\qquad\qquad\qquad\ \textsc{then}\ \mathit{UserOnChainBalance}(u) \\
&\qquad\qquad\qquad\ \textsc{else}\ \ 0] \\
&\quad \land\ \mathit{UserChannelBalance}' = [u \in \textsc{domain}\ \mathit{UserChannelBalance} \mapsto \\
&\qquad\qquad\ \textsc{if}\ \mathit{UserHonest}[u] \\
&\qquad\qquad\ \textsc{then}\ 0 \\
&\qquad\qquad\ \textsc{else}\ \ \mathit{UserChannelBalance}[u]]
\end{aligned}
$$

Figure 3.22.: The action *WithdrawBalancesAfterChannelClosed* is enabled if *LedgerTime* is smaller than a constant *TERMINATION*. The constant is used to note that the system has terminated and no further non-stuttering steps are allowed. The action is only enabled if all channels are in the state `closed`. The action sets all honest user's channel balances to 0 and sets all honest user's external balances to the balance that each user has on the blockchain. This balance on the blockchain is determined using the operator *UserOnChainBalance*. This figure represents a simplified excerpt of the action's actual definition.

set is defined in *SpecificationI* as the union of sets that are defined by operators of the modules *PaymentChannelUser* and *HTLCUser* that are also named *TimeBounds*. The action *AdvanceLedgerTimeI* adds two additional conjuncts to the action *AdvanceLedgerTime*. The first additional conjunct updates the value of the variable *TxAge* which is a function that maps the id of each transaction in the ledger to the age of the transaction. The age of a transaction is increased by the same interval as the variable *LedgerTime*. The second additional conjunct implements urgency for the age of transactions: Time cannot advance if an action must be taken before a transaction's age increases.

3.3.5.2. WithdrawBalancesAfterChannelClosed

The liveness property that we present below in Section 3.3.5.4 ensures that each behavior of *SpecificationI* is terminated by a step of the action *WithdrawBalancesAfterChannelClosed* (see Fig. 3.22). The action is only enabled if the value of *LedgerTime* is below a constant *TERMINATION* and sets the new value of *LedgerTime* to this constant indicating that no further steps are possible. The action sets the channel balances of all honest users to 0 and sets the external balances of all honest users to the balances that these users actually have on the blockchain. Note that the new value of a user's external balance is not set to the current value of the user's channel balance. Instead, the value of the external balance is calculated by the operator *UserOnChainBalance* which calculates a user's balance based on the outputs in the ledger that are solely spendable the given user. This approach ensures that users actually have access to the funds that they should have access to. The balances of dishonest users are not updated because it is not required by the security property that dishonest users withdraw their balance. For a payment channel network to be secure and

correct, it suffices if all honest users successfully withdraw their correct balance. This implies that dishonest users cannot withdraw more than their correct balance.

3.3.5.3. Adversarial Behavior

The formalization models that users can be adversarial. In principle, there are four types of adversarial behavior: (1) Omitting sending a message to another user. (2) Omitting publishing a transaction on the blockchain. (3) Sending a message to another user with arbitrary content. (4) Publishing transactions on the blockchain other than specified by the protocol.

The adversary model for the TLA$^+$ formalization permits the adversary to omit sending messages (1) or publishing transactions (2). Such omissions are modeled by an adversary not taking a step for sending a message or publishing a transaction that is enabled. In Section 3.3.5.4, we describe that we explicitly exclude dishonest users from a liveness condition that enforces that users take a step of an action A if the action A is continuously enabled.

In addition, the adversary is allowed to construct and publish certain transactions other than specified by the protocol (4). Transactions published by an adversary are relevant only if they spend an output of a transaction that is related to the payment channel. In Section 3.3.3.3, we described the two ways that we model to that an adversary publishes transactions other than specified by the protocol: Either the adversary identifies outputs it is able to spend and publishes a new transaction that transfers these outputs to itself or the adversary signs and publishes a transaction for which it has already obtained the counterparty's signature, such as an outdated commitment transaction.

We do not model the adversary as sending messages with arbitrary content (3), as doing so would substantially increase the complexity of the specification. In practice, the impact of the adversary sending messages with arbitrary content is limited because users validate every received message. Messages that have an invalid payload or that are received at an inappropriate point in the protocol execution are ignored. To check the verification of messages, the formalization explicitly models the validation of each received message and its payload including, for example, the validation of signatures and preimages.

In our specification, any user may behave adversarially. As described above, the adversary model is strictly stronger than a crash fault model, yet weaker than a fully Byzantine fault model. In particular, adversarial users may not only crash but may also publish blockchain transactions that deviate from those prescribed by the Lightning protocol. However, we do not model arbitrary adversarial behavior. A further limitation of our adversary model is that we do not allow information sharing between adversarial users which would correspond to a single adversarial entity controlling multiple users. This restriction simplifies the verification of the specification because channels would be less independent if adversarial users would cooperate across channels. We consider it a challenge for future work to extend and verify the specification with a broader adversary model.

3.3.5.4. Liveness

The TLA$^+$ specification of Lightning contains as liveness part a weak fairness condition. Weak fairness of an action specifies that a step of the action must be taken if the action is continuously enabled. The action used for the weak fairness is called *NextIFair* and is defined so that it holds that $NextIFair \Rightarrow NextI$ because each subaction of *NextIFair* is also a subaction of *NextI*.

The subactions of *NextIFair* include *AdvanceLedgerTimeI* and *WithdrawBalancesAfterChannelClosed*, i.e. time advances unless prevented by a time bound and the final update of balances is performed when it is possible. The modules *PaymentChannelUser* and *HTLCUser* specify an action *NextFair* that is a subaction of *NextIFair*. The actions *NextFair* of the two modules include, for an honest user, all actions of the module, i.e. honest users must take a protocol step if the respective action is continuously enabled. This models that honest users do not stop the execution of the protocol but participate in the protocol until their channels are closed.

For dishonest users, the actions *NextFair* of the modules *PaymentChannelUser* and *HTLCUser* include only the actions that receive messages or read from the blockchain but not the actions that send messages or publish transactions (we discuss exceptions to this below). This models that dishonest users update their local information when possible but dishonest users might stop executing the protocol.

To simplify the specification, the module *PaymentChannelUser* makes a few exceptions to this approach: First, the actions to open a payment channel are included in *NextFair* even for dishonest users. This prevents uninteresting behaviors from being explored during model checking because behaviors that do not include a payment channel being opened do not affect the blockchain and, thus, they do not affect any user balances.

Second, when abstracting the protocol modeled in *SpecificationI*, we will assume that both users finally have their *State* variable set to `closed`. However, following the approach described above, dishonest users might not advance their *State* variable to `closed`. To implement that both users finally have their *State* variable set to `closed` without restricting the adversary, a dishonest user is forced to take steps only after the other has already terminated. Because the other user has already terminated, this does not give an advantage to the other user.

A third exception is a very specific scenario in which liveness of dishonest users is required: Assume that an HTLC is part of a commitment transaction that has been honestly published on-chain, i.e. the commitment transaction cannot be revoked. The timelock of the HTLC has already passed. The user for whom the HTLC is incoming is honest but does not have the preimage. The user for whom the HTLC is outgoing is dishonest. Now, the dishonest user could spend the HTLC output because the HTLC has already timed out. However, if the dishonest user never spends the HTLC output, the honest user remains in a state in which the HTLC might be resolved in two different ways: Either the dishonest user spends the HTLC output on-chain or the honest user might receive the preimage for the HTLC and spend the HTLC output. Because we specify that users terminate only

if they know how all HTLCs have been resolved, this situation prevents honest users from terminating. This situation is resolved in the TLA$^+$ specification by detecting this situation and requiring a dishonest user to publish a transaction on-chain that times out the HTLC so that the dishonest user retrieves the HTLC's amount. This exception is only required because of the specific termination condition in our formalization. An alternative approach to this exception would be to change the protocol so that the honest user marks the HTLC as timed out although the HTLC has not actually been timed out on-chain.

3.3.6. Relation of Lightning Specification to Security Property

To given an impression how the steps in *SpecificationI* are mapped to steps of *SpecificationV*, an example is shown in Table 3.1. In the trace shown in the table, a payment channel is opened between two users with ids 1 and 2. The channel is funded by user 1 and the step in which user 1 publishes the funding transaction is mapped to a step of *Deposit* for user 1 in *SpecificationV*. After a payment is initiated and user 2 has received the signature for the new commitment transaction that contains the HTLC for the payment, user 2 closes the channel by publishing this new commitment transaction. User 2 fulfills the HTLC on the blockchain by publishing the respective HTLC success transaction. This step is mapped to a step of *Pay* for user 2 in *SpecificationV*. The step in which user 1 notes the preimage on the blockchain is mapped to a step of *Pay* for user 1 in *SpecificationV*. Both steps of *Pay* of the module *IdealUser* fulfill the specification of *IdealPayments!Pay* because the receiver of the payment performs the *Pay* step before the sender of the payment. Finally, the step of $WithdrawBalancesAfterChannelClosed$ is mapped to a step of *Withdraw* for both users.

3.3.7. Conclusion

This chapter presented a TLA$^+$ formalization for a security property for Lightning that formalizes the property that each honest user eventually receives the user's correct balance and a formalization of Lightning. For formalizing Lightning in TLA$^+$, we used the official specification of Lightning, the BOLTs, as a basis. To ensure that the formalization of Lightning captures the behavior of Lightning as closely as possible, the formalization follows the structure of the official specification. While the BOLTs provide a detailed specification of the message exchange between clients, the specification of how a client should interact with the blockchain by observing and publishing transactions is less detailed and leaves more room for interpretation. Thus, our process of formalizing the protocol was not straightforward but a process of iteratively specifying actions of the protocol and checking whether the induced behaviors correspond to the ideas behind Lightning. In the following chapter, we present our approach for model checking that the TLA$^+$ formalization of Lightning implements the formalized security property.

Step of *SpecificationI*	Step of *SpecificationV*
PCU(1,1)!SendOpenChannel	
PCU(1,2)!SendAcceptChannel	
PCU(1,1)!CreateFundingTransaction	
PCU(1,1)!SendSignedFirstCommitTransaction	
PCU(1,2)!ReplyWithFirstCommitTransaction	
PCU(1,1)!ReceiveCommitTransaction	
HU(1,1)!RequestInvoice	
PCU(1,1)!PublishFundingTransaction	IdealUser(1)!Deposit
PCU(1,1)!SendNewRevocationKey	
PCU(1,2)!NoteThatFundingTransactionPublished	
PCU(1,2)!SendNewRevocationKey	
PCU(1,1)!ReceiveNewRevocationKey	
PCU(1,2)!ReceiveNewRevocationKey	
HU(1,2)!GenerateAndSendPaymentHash	
HU(1,1)!ReceivePaymentHash	
HU(1,1)!AddAndSendOutgoingHTLC	
PCU(1,1)!SendSignedCommitment	
HU(1,2)!ReceiveUpdateAddHTLC	
PCU(1,2)!ReceiveSignedCommitment	
PCU(1,2)!CloseChannel	
PCU(1,2)!ResolveHTLCAfterClose	IdealPayments!Pay, IdealUser(2)!Pay
PCU(1,1)!NoteThatOtherUserClosedHonestly	
PCU(1,1)!NoteThatHTLCFulfilledOnChain	IdealPayments!Pay, IdealUser(1)!Pay
PCU(1,1)!Terminate	
PCU(1,2)!Terminate	
WithdrawBalancesAfterChannelClosed	IdealUser(1)!Withdraw, IdealUser(2)!Withdraw

Table 3.1.: Trace of an execution in which a channel is opened and a payment is made that is resolved on the blockchain. The left column shows the steps of *SpecificationI* while the right column shows the matching steps of *SpecificationV*. In this trace, there is one channel with id 1 (first parameter of PCU and HU) and there are two users with ids 1 and 2 (second parameter of PCU and HU). *SpecificationV* contains the module *IdealPayments* and the module *IdealUser* parameterized for each user. The right column shows which action of these modules describes each step. For better readability, we omit stuttering steps; if the right column is empty, the step of *SpecificationI* is mapped to a stuttering step in all three modules of *SpecificationV*.

4. Model Checking the Security of a Payment Channel Network

After we presented our TLA$^+$ specification of Lightning and a specification of the security property in the previous chapter, in this chapter, we describe our approach for model checking that the TLA$^+$ specification of Lightning fulfills the security property. The specification's state space, however, is so large that directly model checking the TLA$^+$ specification is practically infeasible. To make model checking feasible, we construct a stepwise refinement from the specification of Lightning over three intermediate abstractions to the security property (see Fig. 4.1). With this approach, we address our research question of how to verify that a payment channel network fulfills its security properties.

We refer to the specification of Lightning as presented in the previous chapter as *SpecificationI*. Each payment channel is modeled in *SpecificationI* on two layers: the enforcement layer and the payment layer. Based on *SpecificationI*, we create *SpecificationII* that abstracts the blockchain by replacing the module that enforces the state of a channel on the blockchain by an abstract enforcement module. The abstract enforcement module also abstracts the intra-channel communication on the enforcement level by updating the variables of both users in one step. On the payment layer, HTLCs are used to make payments between users in a payment channel network. In *SpecificationIII*, we further abstract payment channels by abstracting also the payment layer. In this specification, each payment channel is modeled by a single instance of a module that models the entire payment channel and has access to the variables of the two users connected by the payment channel. Information is exchanged between payment channels by passing preimages from one channel to another and the channels are synchronized by a global clock. In *SpecificationIV*, this inter-channel communication is abstracted by modeling payment channels with access to the variables of other payment channels. This allows for a more abstract definition of *SpecificationIV* that does not model preimages or time. The security property as presented in the previous chapter is *SpecificationV*, which is the highest abstraction in this hierarchy in which the payment channel network is abstracted and the balances of users are represented simply as a list of balances. In this chapter, we present the intermediate specifications in more detail and describe how we use proofs and model checking to verify the correctness of each abstraction.

To verify the abstractions, we define mappings between these specifications that determine how a state of a lower-level specification is mapped to a state of a higher-level specification. By showing that these mappings are refinement mappings, it can be proven for each pair of subsequent specifications in the stepwise refinement that the lower-level specification is an

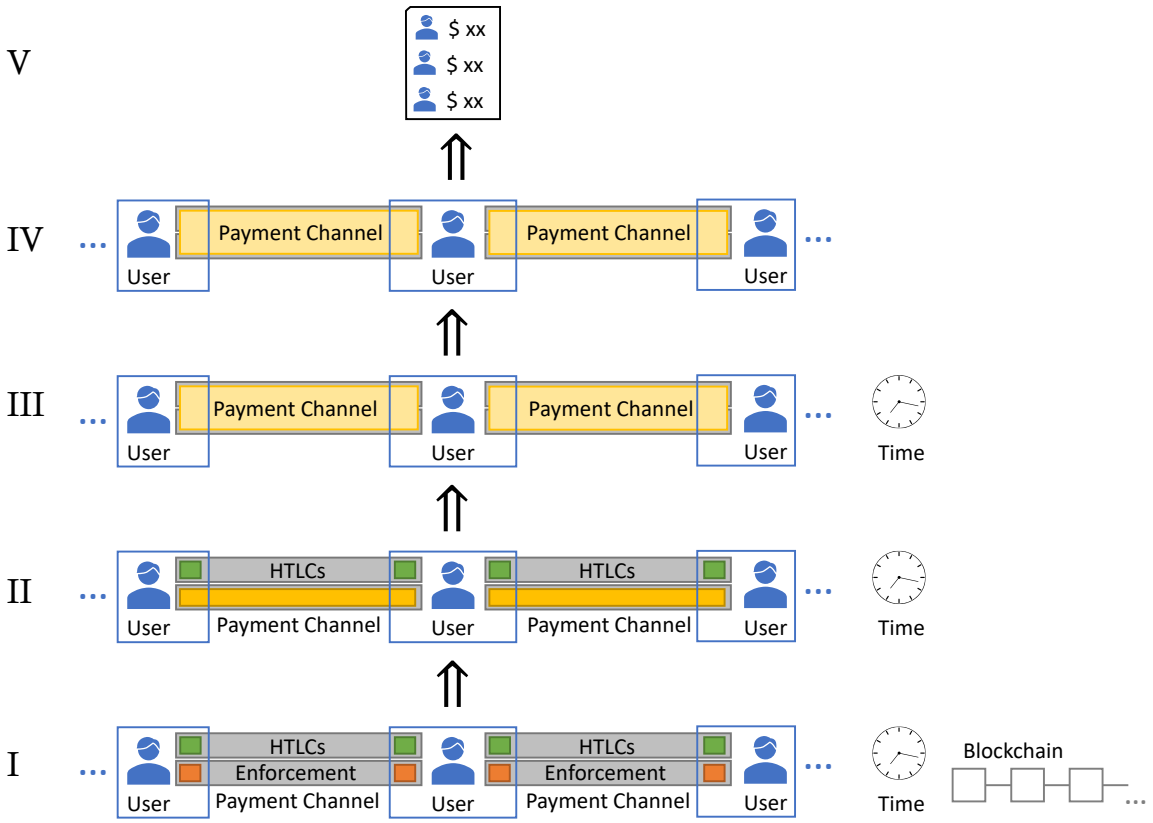

Figure 4.1.: Structure of the stepwise refinement to show that the Lightning protocol specified in *SpecificationI* refines the security property specified as *SpecificationV* in Fig. 3.2. The Roman numerals refer to the specifications.

implementation of the higher-level specification. While a mechanically verified proof for each abstraction would be desired, the complexity of the specifications and the refinement mappings is too high to create such a proof with currently available tools and reasonable effort. Therefore, our approach is to verify the abstractions in the stepwise refinement using model checking. Because the state space of most specifications is too large for direct model checking, we use two main ideas to facilitate the use of model checking:

Time Abstraction We abstract the model of time in a specification to reduce the specification's state space. This abstraction reduces the number of states that only differ in their value of time but are otherwise equivalent.

Decomposition of a Payment Channel Network Into Single Channels To show that a lower-level specification implements a higher-level specification, we decompose the network of payment channels modeled in the lower-level and the higher-level specification into specifications that model only a single payment channel. We show by model checking that it holds for a single channel that the lower-level specification implements the higher-level specification. Based on this result, we prove that the lower-level specification implements the higher-level specification because the higher-level specification is a composition of single channels specified by the higher-level specification.

As the previous chapter, this chapter is based on [GH26a]. The chapter is organized as follows. We first present the approach for reducing a specification's state space by optimizing the model of time in Section 4.1. In this section, we define a general real-time specification in TLA$^+$ and a corresponding time-optimized specification that possibly has a smaller state space. We prove that the time-optimized specification abstracts the general real-time specification. In the following sections, we will apply this general result for our TLA$^+$ specifications. As the result is not specific to our use-case of Lightning, it is of general interest and can be applied to other real-time specifications in TLA$^+$. In Section 4.2, we introduce the approach to show an abstraction by decomposing a specification of a payment channel network into specifications of a single channel.

We show in Section 4.3 that *SpecificationI*, our specification of Lightning, implements *SpecificationII* in which the enforcement layer of payment channels is abstracted. For this abstraction, we apply the time abstraction and the decomposition approach. In Section 4.4, we show that *SpecificationII* implements *SpecificationIII* in which payment channels are abstracted. For this abstraction, we again apply the time abstraction and the decomposition approach. In Section 4.5, we show that *SpecificationIII* implements *SpecificationIV* in which the communication between payment channels is abstracted. For this abstraction, we apply the time abstraction and then check the abstraction using model checking. In Section 4.6, we show by model checking that *SpecificationIV* implements *SpecificationV* which abstracts from the payment channel network and models the security property that the balances of users are consistent with payment states. We present our model checking results in Section 4.7 where we model check the steps of the stepwise refinement for scenarios with one payment over up to six hops and for scenarios with two sequential and two parallel payments. We discuss our approach in Section 4.8 and conclude in Section 4.9.

4.1. Time Abstraction

Specifications such as the TLA$^+$ specification of Lightning as specified in *SpecificationI* are too complex for direct model checking due to the large number of reachable states. A major contributor to this large number of states is the explicit modeling of time. In *SpecificationI*, many states differ only in their specific time values yet their different time values allow for the same behaviors starting from these states. This observation is described by the concept of simulation: A state s_1 is simulated by a state s_2 if, for every behavior that starts in state s_1, there exists a matching behavior of steps of the same actions starting at state s_2. Consequently, to consider all behaviors of a system during model checking, the state s_1 is redundant and it suffices to consider only behaviors starting in the state s_2.

The specification of Lightning in *SpecificationI* is a real-time specification. A real-time specification is the specification of a real-time system which is a system that depends on time. A real-time specification includes clocks that model the passage of time. In our setting, clocks have integer values and can be increased in integer steps. We will define real-time specifications in TLA$^+$ in Section 4.1.1. The time flow of a real-time system is visualized in Fig. 4.2: A clock starts at an initial value and each step that increases

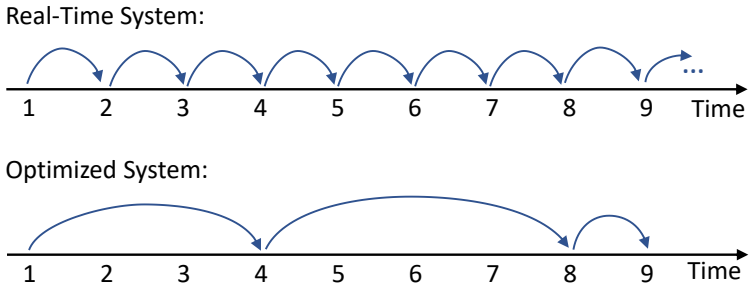

Figure 4.2.: Possible time flows of an exemplary real-time system and the corresponding time-optimized system. For simplicity, the example uses only a single clock. The blue arrows indicate steps of the action *AdvanceTime*. In the real-time system, time increases in arbitrary integer steps. The figure shows behaviors of both systems that contain only steps that increase the clock by the smallest possible steps. In the real-time system, the smallest possible step is a step of a single time unit. In the time-optimized system, time must increase in larger steps. We describe later how the size of the smallest possible steps in the time-optimized system is determined.

time increases the clock's value by an arbitrary integer. Thus, each positive integer is a possible clock value. In Section 4.1.3, we define a corresponding time-optimized system in which redundant states are skipped and time must advance in larger steps (see Fig. 4.2)[1]. We prove in Section 4.1.5 for a general real-time system that the definition of the time-optimized system is an abstraction of the real-time system. To technically facilitate the proof, we introduce a further specification as an extension of the real-time system's specification in Section 4.1.4. The extended real-time specification adds to the real-time system's specification, for each clock, an auxiliary variable that stores the clock's value in the time-optimized system. We prove that the general real-time system implements the time-optimized system (stated by Theorem 1) by first showing that the real-time system implements the extended real-time system (Lemma 1) and then that the extended real-time system implements the time-optimized system (Lemma 3). The proof that a general real-time system implements the corresponding time-optimized system can be applied to any real-time system that is specified according to our definition of real-time systems in Section 4.1.1. The proof relies on a few assumptions about the specification of the real-time system that we define in Sections 4.1.1 and 4.1.3. To apply the proof to a specific real-time system, one has to prove that the specific real-time system's specification fulfills these assumptions. The overview of this section is summarized in Fig. 4.3.

4.1.1. Real-Time Specifications

We start by defining real-time systems in TLA$^+$. The real-time system S to be optimized is defined by an explicit real-time specification $Spec_S$ as $Spec_S = Init_S \wedge \Box[Next_S]_{v_S \cup \mathcal{X}} \wedge$

[1] In the context of Discrete Event Simulation [Law07], an analogy for this idea is to switch from a fixed-increment time advance approach in which time is incremented by the smallest possible unit to a next-event time advance approach.

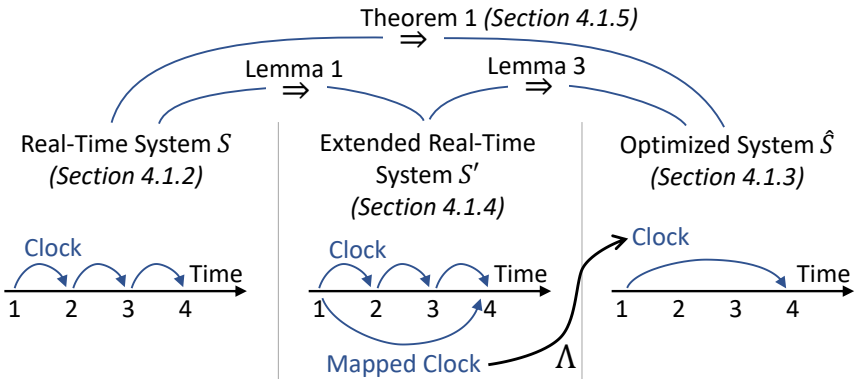

Figure 4.3.: Overview of Section 4.1 about the abstraction step to optimize time. Starting from a real-time system S (left) in which time can increase by steps of a single time unit, we define a time-optimized system $\hat{S}$ (right) in which time must increase in larger steps. To facilitate the proof, we define an extended real-time system S' (center) that contains for each clock a mapped clock variable. The value of each mapped clock variable is the value to which the corresponding clock in the real-time system is mapped to in the time-optimized system. Thus, each state of the extended system S' can be mapped to a state of the time-optimized system $\hat{S}$ by setting the value of each clock to the value of the corresponding mapped clock variable (mapping Λ). We prove that the real-time system S implements the time-optimized system $\hat{S}$ (Theorem 1) by proving that the real-time system S implements the extended system S' (Lemma 1) and that the extended real-time system S' implements the time-optimized system $\hat{S}$ (Lemma 3).

$Liveness_S$ with the set of variables v_S and the set of clocks $\mathcal{X}$. The system S has the state space Σ_S which contains all reachable states of the specification. We use Σ to refer to the state space of all *possible* states using the variables v_S, i.e. $\Sigma_S \subseteq \Sigma$. In this definition and our notation, we distinguish variables from clocks, in TLA$^+$ however, clocks are modeled as variables. The $Next_S$ action of the specification $Spec_S$ is a disjunction of an internal next action $InnerNext$ and an $AdvanceTime_S$ action: $Next_S = InnerNext_S \vee AdvanceTime_S$. Below, we define how the $AdvanceTime_S$ action is specified. The action $InnerNext_S$ must leave the value of clocks unchanged except that inactive clocks may be activated by updating a clock's value from $\perp$ to the clock's initial value (will be explained below). $Liveness_S$ is the conjunction of formulas of the form $WF_{v_S}(A)$ (weak fairness) for subactions A of $Next_S$ and $SF_{v_S}(A)$ (strong fairness) for subactions A of $Next_S$[2].

To give an overview of the specification of $Spec_S$, Fig. 4.4 shows the composition of the definitions that we give here. Because clocks are modeled as variables and TLA$^+$ does not allow for quantification over sets of variables ($\forall x \in X$), the specification in Fig. 4.4 is not written in correct TLA$^+$ but instead uses a pseudo form of TLA$^+$ so that the specification corresponds to the actual definitions.

We assume that the specification has the following properties: The clocks $\mathcal{X}$ are modeled as variables that are not externally visible[3] and the set of variables v_S does not contain the clocks in $\mathcal{X}$, i.e. $\mathcal{X} \cap v_S = \emptyset$. All values of a clock $x \in \mathcal{X}$ are elements of the set $\mathbb{N}_0$.

[2] Weak fairness for action A requires that a step of A must occur if the action A is continuously enabled. Strong fairness for action A requires that a step of A must occur if the action A is enabled infinitely often.
[3] If a clock was externally visible, we could not abstract the behavior of the clock.

$Init \triangleq$ initially,
$\quad \land \forall x \in X :$ all clocks
$\quad\quad \lor x = I[x]$ are set to their initial value or
$\quad\quad \lor x = INACTIVE$ are set to INACTIVE

$AdvanceTime \triangleq$
$\quad \land$ UNCHANGED v all non-clock variables are unchanged
$\quad \land \forall x \in X :$ for all clocks x in X
$\quad\quad$ IF $x = INACTIVE$ if the clock is inactive
$\quad\quad$ THEN $x' = INACTIVE$ the clock stays inactive
$\quad\quad$ ELSE $\exists n \in Nat : x' = x \land x' > x$ else, the clock increases to any natural number
$\quad \land \forall x \in X :$ for all clocks x in X
$\quad\quad x \neq INACTIVE \implies$ if the clock is active
$\quad\quad\quad \forall b \in TimeBounds(x) : x \leq b \implies x' \leq b$ no time bound is skipped

$InnerNext \triangleq$
$\quad \land \ldots$
$\quad \land \forall x \in X :$
$\quad\quad \land$ IF $x = INACTIVE$
$\quad\quad\quad$ THEN $x' = I[x] \lor$ UNCHANGED x inactive clocks are activated or unchanged
$\quad\quad\quad$ ELSE UNCHANGED x active clocks are unchanged

$Next \triangleq InnerNext \lor AdvanceTime$

$Spec \triangleq Init \land \Box[Next]_{\langle v, X \rangle} \land Liveness$

Figure 4.4.: Pseudo-TLA$^+$ specification of a real-time system to give an overview of the definitions in Section 4.1.1. The specification does not contain the subscripts for the system S ($_S$) because these are not allowed in TLA$^+$ syntax. We use the notation $I[x]$ for the initial value of clock x which we defined as the mapping I^x. Similarly, we use the notation $TimeBounds(x)$ for the set $B^x(s)$ where s is the current state.

All clocks $x \in \mathcal{X}$ have an initial value $I^x \in \mathbb{N}_0$. Consistently across all initial states, each clock $x \in \mathcal{X}$ is initially assigned either the clock's initial value I^x or $\perp$ which indicates that the clock is inactive. A clock can be activated by an action by assigning the clock its initial value. Once activated, clocks cannot be deactivated. The action $AdvanceTime_S$ advances the values of all active clocks $x \in \mathcal{X}$ by an arbitrary integer.

A real-time specification may specify that steps are urgent in the sense that they must occur before a certain point in time. This is implemented using time bounds which are points in time that a clock may not increase beyond. Time bounds are defined for each clock $x \in \mathcal{X}$ by a mapping B^x that maps a state to a set of natural numbers. In a state s, the $AdvanceTime_S$ action is disabled if any clock x has a value that is equal to one of the time bounds in the set $B^x(s)$. We illustrate the effect of time bounds below in Example 4.1.

We formally define the action $AdvanceTime_S$. For this definition, we first define an equivalence relation that defines two states as varequal if all of their variables, but not necessarily their clocks, have the same values. We will use this equivalence relation in several definitions below.

Definition 3 (Varequality of states, $\approxeq$)**.** Two states s, t are varequal, written $s \approxeq t$ if and only if both states have the same set of variables and it holds for all variables v of the states s and t that $s.v = t.v$. The clock values in the states s and t may, however, be different.

Using the definition of varequality, we define the action $AdvanceTime_S$. The definition is first given formally and then explained informally.

Definition 4 ($AdvanceTime_S$)**.** For two states $s, t \in \Sigma$, it holds that $s \xrightarrow{AdvanceTime_S} t$ if and only if

$$
\begin{aligned}
& s \approxeq t \\
& \wedge \forall x \in \mathcal{X} : \begin{cases} t.x = \bot & \text{if } s.x = \bot \\ \exists n \in \mathbb{N} : t.x = n \wedge t.x > s.x & \text{else} \end{cases} \\
& \wedge \forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b
\end{aligned}
$$

Rephrased informally, the definition of $AdvanceTime_S$ defines that a step from state s to state t is an $AdvanceTime_S$ step if and only if three conditions are fulfilled:

- The states s and t are varequal, i.e. all variables have the same values ($s \approxeq t$).
- For each clock, it holds that, if the clock is active, the clock's value must increase ($t.x > s.x$) to an arbitrary integer ($\exists n \in \mathbb{N} : t.x = n$). Otherwise, if the clock is inactive, the clock stays inactive ($t.x = \bot$).
- For each active clock x, it holds that the clock's value does not pass a time bound in the set of time bounds $B^x(s)$ of clock x in state s, i.e. if the clock's value was below the time bound in state s then the clock's value is still below the time bound in state t.

Example 4.1 (Time Bounds)**.** Consider an exemplary real-time system with a single clock x and a time bound at time 7. Assume that in a state s_1 the value of the clock x equals 1 and $B^x(s_1) = \{7\}$, i.e., there is a time bound that prevents the clock x to advance beyond 7. An exemplary behavior of this system is sketched in Fig. 4.5. Starting at state s_0, after a step of $AdvanceTime_S$, a step of $InnerNext_S$, and several $AdvanceTime_S$ steps more, a state s_7 is reached in which the clock x has the value 7. Thereby, the clock x has reached the value 7 that is a time bound in state s_7. In state s_7, an $AdvanceTime_S$ step is not possible because the second condition of the action $AdvanceTime_S$ requires that, for a step from state s_7 to a state s_8, it must hold that $s_8.x > s_7.x$ but the third condition requires that it holds for the time bound $b = 7$ that $s_7 \leq b \Rightarrow s_8 \leq b$ which is not fulfilled. Therefore, only a step of $InnerNext_S$ is possible in state s_7. After two steps of $InnerNext_S$, a step t_7 might be reached so that $B^x(t_7) = \{\}$ and time can advance further.

The proof of Theorem 1 relies on the assumption that the value of the time bounds $B^x(s)$ for a state s of the state space Σ of the real-time system is independent of the clock values in state s. For a concrete specification that defines B^x, it must be proven that this assumption as formalized in Assumption 4.1 is met. Informally, the assumption requires that for every

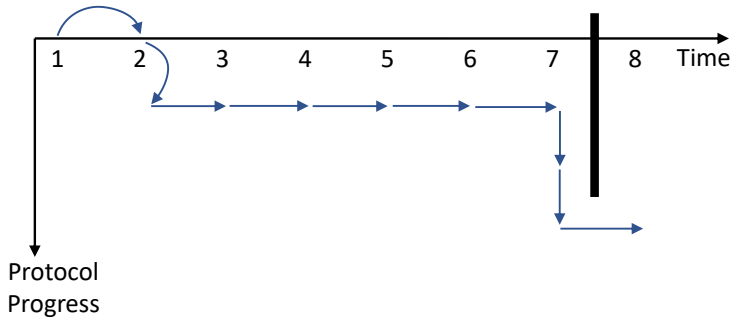

Figure 4.5.: Behavior of an exemplary real-time specification with a time bound and a single clock. Blue arrows from left to right represent steps of $AdvanceTime_S$ that increase the value of the clock. Blue arrows from top to bottom represent steps of other subactions of $Next_S$. The shown behavior takes multiple steps of $AdvanceTime_S$ and a step of $InnerNext_S$ until the clock reaches the time value 7 at which a time bound exists. Thus, at this point in time, time cannot advance further. After two steps of $Next_S$ are taken, a state is reached at which the time bound does not exist anymore and the action $AdvanceTime_S$ is enabled and time can advance further.

state $s_1 \in \Sigma$ and every state s_2 that is equal to state s_1 except for the value of a clock $y \in \mathcal{X}$ it holds that the time bounds for a clock x in the states s_1 and s_2 are equal.

Assumption 4.1. *For all clocks $x, y \in \mathcal{X}$, all $d \in \mathbb{N}_0$, all states $s_1 \in \Sigma$, and all states s_2 that are equal to state s_1 except that $s_2.y = d$ it must hold that:*

$$B^x(s_1) = B^x(s_2)$$

4.1.2. Optimization Idea

In the following, we explain the idea of the optimization of time in more detail. To this end, we consider a general real-time system specified in TLA$^+$ as defined in Section 4.1.1 and explain how we define a time-optimized system that is implemented by the real-time system while potentially having a smaller state space. We demonstrate the abstract ideas at a simple concrete real-time specification that is inspired by the Lightning setting.

Our goal is to define a time-optimized specification so that it is an abstraction of a real-time specification. The time-optimized specification is an abstraction of the real-time specification if and only if there exists a refinement mapping that maps every behavior of the real-time system to a behavior of the corresponding time-optimized system [AL91]. To ensure the existence of such a refinement mapping by construction, we define the time-optimized specification so that every state of the real-time system is simulated by a state of the time-optimized system. By definition of simulation, this definition of the time-optimized specification implies the existence of a refinement mapping from the real-time specification to the time-optimized specification because each behavior of the real-time system that starts in state s can be mapped to a behavior of the time-optimized system that starts in a state that simulates the state s. In principle, it follows from this construction that the real-time specification implements the time-optimized specification. However,

because we describe the construction approach in parts informally, we explicitly prove that the real-time specification implements the time-optimized specification by specifying a refinement mapping from the real-time specification to the time-optimized specification and proving that the refinement mapping is correct.

The variables of a system can be divided into internally visible and externally visible variables: Externally visible variables are the variables of a system that are visible to an external observer while internally visible are hidden from an external observer. In the case of Lightning, the externally visible variables are those variables that are included in the security property, e.g., the variables that model payments and balances. To retain the externally visible behavior of a system during abstraction, a refinement mapping may change the values of internally visible variables but must not change the values of externally visible variables. To ensure that the refinement mapping from the real-time specification to the time-optimized specification that is implied by the simulation relation does not change external variables, we define simulation in the following Definition 5 so that a state t *simulates* another state s only if the values of the variables of both states are equal but not necessarily the values of their clocks. This requires that clocks are modeled as internal variables, as it is the case for a real-time system as defined in Section 4.1.1, but allows for all non-clock variables to be externally visible.

A formal definition of simulation follows in Definition 5. We define *simulation* as a relation between two states. The second condition ensures that every behavior starting in state s_1 can be mapped to a behavior starting in state s_2 so that the steps in both behaviors are described by the same sequence of actions. The condition can be rephrased in prose as: if there exists a state t_1 so that there exists an action A so that the step from state s_1 to state t_1 is an A step, then there exists a state t_2 so that the step from state s_2 to state t_2 is an A step and the state t_1 is simulated by the state t_2. Thus, the definition of simulation is recursive. A base case in which the recursion stops is reached when a state t_1 is reached in which no actions are enabled or when two states t_1 and t_2 are reached so that $t_1 = t_2$.

Definition 5 (Simulation, $\rightsquigarrow$)**.** A state s_1 is simulated by a state s_2, written $s_1 \rightsquigarrow s_2$, if

1. $s_1 \cong s_2$, i.e., the states s_1 and s_2 are varequal, and,
2. if there exists a state t_1 so that $s_1 \xrightarrow{A} t_1$ for an action A, then there exists a state t_2 so that $s_2 \xrightarrow{A} t_2$ and $t_1 \rightsquigarrow t_2$.

We also say that the state s_2 simulates the state s_1.

Remark. It follows from Definition 5 that simulation is reflexive, i.e., any state s simulates itself ($s \rightsquigarrow s$), and transitive, i.e., if it holds for the states s, t, u that the state s is simulated by the state t and the state t is simulated by the state u, then the state s is simulated by the state u (formally: $s \rightsquigarrow t \land t \rightsquigarrow u \Rightarrow s \rightsquigarrow u$).

We demonstrate the definition of simulation with an example:

module *SimpleHTLCs*

extends *Integers*

variables *Time*, *HTLCs*

$$
\begin{aligned}
&\mathit{TypeOK} \triangleq \\
&\quad \land \quad \mathit{Time} \in \mathit{Nat} \\
&\quad \land \quad \mathit{HTLCs} \subseteq [\mathit{id} : \mathit{Nat},\ \mathit{timelock} : \mathit{Nat},\ \mathit{state} : \{\text{“NEW”},\ \text{“FULFILLED”},\ \text{“TIMED OUT”}\}] \\
&\mathit{Init} \triangleq \\
&\quad \land\ \mathit{Time} = 1 \\
&\quad \land\ \mathit{HTLCs} = \{[\mathit{id} \mapsto 1,\ \mathit{timelock} \mapsto 10,\ \mathit{state} \mapsto \text{“NEW”}], \\
&\qquad\qquad\quad [\mathit{id} \mapsto 2,\ \mathit{timelock} \mapsto 20,\ \mathit{state} \mapsto \text{“NEW”}]\} \\
&\mathit{AdvanceTime} \triangleq \\
&\quad \land \exists\, n \in \mathit{Nat} : \mathit{Time}' = n \\
&\quad \land\ \mathit{Time}' > \mathit{Time} \\
&\quad \land \text{unchanged } \mathit{HTLCs} \\
&\mathit{Fulfill}(\mathit{htlc}) \triangleq \\
&\quad \land\ \mathit{htlc.state} = \text{“NEW”} \\
&\quad \land\ \mathit{Time} < \mathit{htlc.timelock} \\
&\quad \land\ \mathit{HTLCs}' = (\mathit{HTLCs} \setminus \{\mathit{htlc}\}) \cup \{[\mathit{htlc} \text{ except } !.\mathit{state} = \text{“FULFILLED”}]\} \\
&\quad \land \text{unchanged } \mathit{Time} \\
&\mathit{Timeout}(\mathit{htlc}) \triangleq \\
&\quad \land \quad \mathit{htlc.state} = \text{“NEW”} \\
&\quad \land \quad \mathit{Time} \geq \mathit{htlc.timelock} \\
&\quad \land \quad \mathit{HTLCs}' = (\mathit{HTLCs} \setminus \{\mathit{htlc}\}) \cup \{[\mathit{htlc} \text{ except } !.\mathit{state} = \text{“TIMED OUT”}]\} \\
&\quad \land \quad \text{unchanged } \mathit{Time} \\
&\mathit{Next} \triangleq \\
&\quad \lor\ \mathit{AdvanceTime} \\
&\quad \lor \exists\, \mathit{htlc} \in \mathit{HTLCs} : \mathit{Timeout}(\mathit{htlc}) \lor \mathit{Fulfill}(\mathit{htlc}) \\
&\mathit{Spec} \triangleq\ \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{Time},\, \mathit{HTLCs} \rangle}
\end{aligned}
$$

Figure 4.6.: Exemplary real-time specification that is inspired by the Lightning setting. The specification is further explained in Example 4.2. For the purpose of this example, the *Init* predicate statically initializes the variable *HTLCs* with two HTLCs.

Example 4.2. Consider the example specification in Fig. 4.6 which is a real-time specification that is inspired by the Lightning setting. The specification contains two variables *Time* and *HTLCs*. *Time* is an integer and we interpret it as a clock. The variable *HTLCs* is a set that contains records for HTLCs. Each HTLC record has an id, a timelock, and a state which can either be new or timed out. The *Next* action of the specification has three subactions *AdvanceTime*, *Fulfill*, and *Timeout*. The action *AdvanceTime* increments the value of the variable *Time* to a natural number that is greater than the previous value of *Time*. The action *Fulfill* sets the state of an HTLC to

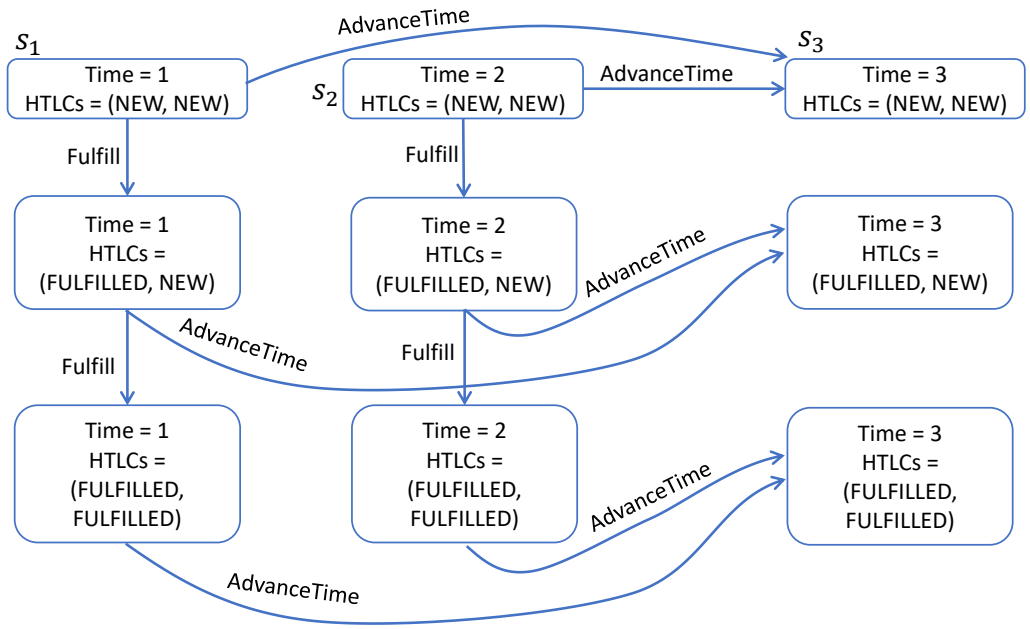

Figure 4.7.: Excerpt of the state space of the specification shown in Fig. 4.6 to show that the state in the top center (s_2) is simulated by the state in the top left (s_1) (see Example 4.2). The figure shows states in boxes that show the values of the clock *Time* and the variable *HTLCs*. The arrows between the states show actions that describe a step between the two connected states. The values of the variable *HTLCs* are abbreviated and only the HTLC's states are shown.

FULFILLED if the value of the variable *Time* is below the HTLC's timelock. The action *Timeout* sets the state of an HTLC to TIMED OUT if the HTLC's timelock has passed. Thus, there are two time-dependent conditions in this specification: One condition in the action *Fulfill* that compares the value of *Time* against an HTLC's timelock and a second condition in the action *Timeout* that checks whether the timelock of an HTLC has passed.

Consider the following two states (see Fig. 4.7 for a visualization): Let state s_1 equal the initial state that is described by the *Init* predicate, i.e., the value of the variable *Time* equals 1 and both HTLCs are in state NEW. Let state s_2 equal state s_1 except that $s_2.Time = 2$. We show that the state s_2 is simulated by the state s_1: By definition, the two states s_1 and s_2 are varequal because they assign the same value to the variable *HTLCs* and *Time* is a clock. To show the second condition of Definition 5, we need to show that for every step starting at state s_2 there is a matching step from state s_1 of the same action and the state reached from state s_2 is simulated by the state reached from state s_1. In the two states s_1 and s_2, only steps of the actions *AdvanceTime* and *Fulfill* are possible. First, we consider the action *AdvanceTime*. A step of the action *AdvanceTime* that advances time in state s_2 (Fig. 4.7 shows an advancement to 3), is also possible starting at state s_1. Because they reach the same state (s_3 in the figure), the state reached from state s_2 is trivially simulated by the state reached from state s_1. Now, we consider the action *Fulfill*. The two HTLCs can be fulfilled in both states s_1 and s_2. Thus, for each step that starts at state s_2 and fulfills an HTLC there exists a step starting at state s_1. The states reached from state s_2 are simulated by the states reached from state s_1 because for each step that fulfills an HTLC there exists a step that fulfills an HTLC and, after both HTLCs have been fulfilled, the only step that is enabled is an *AdvanceTime* step and with such an *AdvanceTime* step, the same state can be reached from both states. This shows that every behavior starting at state s_2 can be mapped to a behavior starting at state s_2 so that both behaviors

consists of steps of the same sequence of actions. It follows that the state s_2 is simulated by the state s_1.

Remark. The definition of simulation differs from the typical definition of simulation [BK08, Definition 7.58] or time-abstracted (bi)simulation [Bou+18, Definition 2] in the aspect that, according to our definition, a state can only simulate another state if the two states are varequal. This ensures that, if a state of a real-time specification is mapped to a state in the time-optimized specification that simulates the state of the real-time specification, the external variables are unchanged which is a necessary property for the real-time specification to be a refinement of the time-optimized specification. Further, our definition of simulation differs from the definition of time-abstracted (bi)simulation [Bou+18, Definition 2] in the aspect that $AdvanceTime$ steps must increase the time, i.e., the value of at least one clock must increase during an *AdvanceTime* step.

As explained above, our idea is to define the time-optimized specification so that each state of the real-time specification is mapped to a state in the time-optimized specification that simulates the state of the real-time specification. Under this constraint, the time-optimized system has the smallest state space if all states of the real-time system that are simulated by the same state s are mapped to the state s. To formalize this idea, we define a *zone* in Definition 9 which is a set of states that are simulated by the same state. The state that simulates all other states of the zone is called a *zone representative*. For technical reasons, we define a zone not simply as the set of *all* states that are simulated by the same state but there are additional properties that states must have to be in the same zone. These additional properties are needed, for example, to achieve that not only every behavior of the real-time system is a behavior of the time-optimized system but also that every behavior of the time-optimized system is also a behavior of the real-time system.[4] To explain the definition of a zone, we start with a first attempt of defining a zone and refine the definition attempts until we have derived at our actual definition of a zone in Definition 9.

Our first attempt for defining a zone is Definition 6 which follows. Informally, Definition 6 defines a zone as the largest set Z of states so that there exist a state $z_1 \in Z$ that is called the zone representative so that all other states of the set Z are simulated by the state z_1.

Definition 6 (First attempt of a definition of a zone)**.** We define a *zone* to be a subset Z of a real-time system's state space Σ_S so that

1. there exists a state $z_1 \in Z$ so that all other states $z_2 \in Z$ are simulated by the state z_1, and
2. there is no subset of the state space Σ_S that fulfills the above conditions and is a proper superset of the set Z.

[4] We need the property that the time-optimized system is still a real-time system for the abstraction of *SpecificationI*^{time} to *SpecificationII* because *SpecificationII* is defined as a real-time system. The refinement mapping from *SpecificationI*^{time} to *SpecificationII* would be more complicated if *SpecificationI*^{time} would not be a real-time system.

A state $z_1 \in Z$ of the zone that simulates all other states $z_2 \in Z$ is called a *zone representative*.

For now, we assume that the time-optimized specification is defined so that each state of the real-time specification is mapped to its zone representative that is defined by Definition 6. Thereby, the time-optimized specification has the following two properties: First, the state space of the time-optimized specification is reduced as much as possible because the time-optimized system's state space does not contain any redundant states. Second, the real-time specification implements the time-optimized specification because each state s of the real-time specification is mapped to the zone representative of state s which is a state that the state s is simulated by.

Remark. By defining a zone using Definition 6, the state space of the time-optimized system is smaller than the state space of the original real-time system if and only if there exists a zone that contains more than a single state or, equivalently, if there exists a state in the real-time system's state space that is simulated by another state.

Remark. Definition 6 implies that all states in a zone are varequal because all states of a zone are simulated by the same state.

For the mapping from the real time specification to the time-optimized specification, we need to find a zone representative for each state. The first attempt for the definition of a zone in Definition 6 implies a time-optimized specification with a minimal state space but the definition does not make it easy to find a zone representative for a given state. To facilitate finding a zone representative, we introduce an additional property for a zone: The zone representative that simulates all states in the zone must have the smallest clock values of all states in the zone. If a real-time system has multiple clocks, this formulation of the property is not well-defined because there might exist zones that contain no state that has for all clocks the smallest clock value. To clearly identify a zone representative for each zone, we restrict zones to sets of states that differ only in the clock value of a specific clock. Then, the state with the smallest clock values is clearly defined.

We summarize these two changes in a second attempt to define a zone in Definition 7.

Definition 7 (Second attempt of a definition of a zone)**.** We define a *zone* to be a subset Z of a real-time system's state space Σ_S so that there exists a clock $x \in \mathcal{X}$ so that

1. in all states in the set Z, the values of all clocks $y \in \mathcal{X}$ so that $y \neq x$ are equal, and
2. the state $z_1 \in Z$ with the lowest clock value of the clock x simulates all other states $z_2 \in Z$, and
3. there is no subset of the state space Σ_S that fulfills the above conditions and is a proper superset of the set Z.

The state $z_1 \in Z$ with the lowest clock value of the clock x (i.e., $\forall z_2 \in Z : z_1.x \leq z_2.x$) is called a *zone representative*. To explicitly name the clock x, a zone is also called an x-zone. The zone representative of an x-zone is an x-zone representative.

Remark. Because zones as defined by Definition 7 can be larger than zones as defined by Definition 6, defining the time-optimized specification so that each state of the real-time specification is mapped to its zone representative according to Definition 7 results in a potentially larger state space of the time-optimized specification compared to a definition of the time-optimized specification based on Definition 6. This smaller reduction of the state space is offset by a technically easier identification of zone representatives.

Remark. By making the definition of a zone in Definition 7 dependent on a specific clock, a specific state of the real-time system is not anymore contained only in a single zone but each state is contained in one zone per clock. Thus, if a specification has multiple clocks, there are multiple zone representatives for each state. Consequently, it is not clear to which zone representative a state of the real-time system is mapped by the refinement mapping from the real-time system to the time-optimized system. For the next paragraphs, we ignore this problem by assuming that there is only a single clock in the specification. We will handle the problem later by adapting how the refinement mapping from the real-time system to the time-optimized system is defined.

The definition of a zone in Definition 7 results in the time-optimized specification not being a real-time specification because Definition 7 allows zones to be non-continuous in the sense as shown in the following example. Remember that we assume for now that the time-optimized specification is defined by mapping each state of the real-time specification to the state's zone representative. The example shows that the problem of Definition 7 is that it leads to a time-optimized specification that contains steps in which time goes backwards. Such a time-optimized system would not be a real-time system.

Example 4.3. Consider a specification with a single clock x and a state s. In this example, we use the notation s_i to refer to a state that is varqual to state s and $s_i.x = i$, i.e., the clock value of clock x in state s_i equals i. According to Definition 7, it could be that the states that are varequal to state s are divided into the following two zones Z_1 and Z_2 if all states s_i where i is an odd number are simulated by state s_1 and all states s_i in which i is an even number are simulated by the state s_2.

$$Z_1 = \{s_1, s_3, s_5, ...\}$$
$$Z_2 = \{s_2, s_4, s_6, ...\}$$

The zone representatives of the zones Z_1 and Z_2 are the states s_1 and s_2 respectively. Consequently, all states s_i would be mapped to the two states s_1 and s_2 in the time-optimized specification. The step $s_1 \rightarrow s_2$ in the real-time system in which the value of the clock x increases from 1 to 2 would be mapped to the step $s_1 \rightarrow s_2$ in the time-optimized system. However, the step $s_2 \rightarrow s_3$ in the real-time system would be mapped to the step $s_2 \rightarrow s_1$ in the time-optimized system which means that, in this step, the value of the clock x is reduced from 2 to 1. By the definition of a real-time specification, such a step is not possible in a real-time specification.

The reason why steps in the real-time system can be mapped to steps in the time-optimized system in which time goes backwards is that behaviors are possible in the real-time system in which a zone is left (e.g., zone Z_1 is left in the step $s_1 \rightarrow s_2$ in Example 4.3) and later

the same zone is entered again (e.g., zone Z_1 is entered again in the step $s_2 \rightarrow s_3$ in Example 4.3). To avoid such steps, we refine the second attempt for the definition of a zone by requiring that zones are continuous in the sense that, given that the states in a zone differ by the values of the clock x and there are two states z_1 and z_2 in the zone whose clock values for the clock x differ by more than a single time unit, then the zone must contain a state that has a clock value that is between the clock values of the two states z_1 and z_2. By including this property, we define the third attempt for the definition of a zone:

Definition 8 (Third attempt of a definition of a zone)**.** We define a *zone* to be a subset Z of a real-time system's state space Σ_S so that there exists a clock $x \in \mathcal{X}$ so that

1. in all states in the set Z, the values of all clocks $y \in \mathcal{X}$ so that $y \neq x$ are equal, and
2. for every two states z_1 and z_2 of the set Z it holds that if $z_1.x - z_2.x \geq 2$ then there is also a state z_3 in the set Z so that $z_1.x > z_3.x > z_2.x$, and
3. the state $z_1 \in Z$ with the lowest clock value of the clock x simulates all other states $z_2 \in Z$, and
4. there is no subset of the state space Σ_S that fulfills the above conditions and is a proper superset of the set Z.

The state $z_1 \in Z$ with the lowest clock value of the clock x (i.e., $\forall z_2 \in Z : z_1.x \leq z_2.x$) is called a *zone representative*. To explicitly name the clock x, a zone is also called an x-zone. The zone representative of an x-zone is an x-zone representative.

By defining the time-optimized specification so that each state of the real-time system is mapped to the state's zone representative according to Definition 8, there is the possibility that an $InnerNext_S$ step of the real-time system is mapped to a step of the time-optimized system in which the values of clocks change. Remember that an $InnerNext_S$ step is a step of the real-time system that is not an $AdvanceTime_S$ step, i.e. the step does not change clock values. Because we want that each behavior of a time-optimized system is also a behavior of a real-time system, an $InnerNext_S$ step in the real-time system should be mapped to a matching step in the time-optimized system in which the clock values are not changed. We demonstrate the problem with the following example.

Example 4.4. Consider a specification with a single clock x. In this example, we again use the notation s_i to refer to a state that is varqual to a state s and $s_i.x = i$. Consider a step $s_3 \xrightarrow{InnerNext_S} t_3$ of the real-time system. Assume that the state s_3 is simulated by the state s_2 and that the state t_3 is simulated by the state t_1. Further assume that the state t_2 is also simulated by the state t_1. Assuming that the states s_2 and t_1 do not simulate any other states, there exist the two zones $Z_1 = \{s_2, s_3\}$ and $Z_2 = \{t_1, t_2, t_3\}$ with the zone representatives s_2 and t_1 respectively. Consequently, the state s_3 of the real-time system is mapped to the state s_2 in the time-optimized system and the state t_3 in the real-time system is mapped to the state t_1. Thus, the step $s_3 \xrightarrow{InnerNext_S} t_3$ of the real-time system is mapped to the step $s_2 \xrightarrow{InnerNext_S} t_1$ in the time-optimized system in which the value of

the clock x changes from 2 to 1. Given that we want that every behavior of the time-optimized system is also a behavior of a real-time system, a clock's value should not change during a step of $InnerNext_S$. Specifically in this example, the step $s_3 \xrightarrow{InnerNext_S} t_3$ of the real-time system should be mapped to the step $s_2 \xrightarrow{InnerNext_S} t_2$ in the time-optimized system so that the value of the clock x is unchanged during the step.

To achieve that steps of $InnerNext_S$ in the real-time system are mapped to steps in the time-optimized system in which the clock values remain constant, the following two aspects are adapted: First, we adapt the definition of the refinement mapping from the real-time system to the time-optimized system so that the refinement mapping does not necessarily map a state s of the real-time system to the state's zone representative but possibly to another state that is in the same zone as the state s. Second, because the adapted refinement mapping maps a state s possibly to a state $\hat{s}$ that is not a zone representative, the definition of a zone must guarantee that the state $\hat{s}$ simulates the state s.

The refinement mapping from the real-time system to the time-optimized system should map a state s of the real-time system to a state $\hat{s}$ with the following properties. We will later ensure that the state $\hat{s}$ simulates the state s. To express the properties, we use the term predecessor of a state. By the predecessor of the state s, we mean the state s_0 from which the state s was reached by a step $s_0 \rightarrow s$ if the state s is not an initial state.

- The state $\hat{s}$ is varequal to the state s.
- For each clock $x \in \mathcal{X}$, the clock value in state $\hat{s}$ is set to the maximum of the following two values:
  - the clock value of the x-zone representative of state s and
  - if the state s is not an initial state, the value that the clock x was mapped to by the refinement mapping for the predecessor of state s

By setting the value of a clock at least to the value that the clock was mapped to for a state's predecessor, it is ensured that a clock's value does not decrease when a step of $InnerNext_S$ is mapped to the time-optimized system. These properties of the refinement mapping depend on the values that the clock values of a state's predecessor were mapped to. Therefore, this refinement mapping can only be explicitly defined by adding auxiliary variables to the real-time specification (see [AL91]). To this end, we define an extended real-time specification in Section 4.1.4 that equals the real-time specification except that auxiliary variables are added. The extended real-time specification has an auxiliary variable for each clock $x \in \mathcal{X}$. These auxiliary variables store the value that the clock x is mapped to. During a step of $AdvanceTime_S$ from state s to state t, the values of the auxiliary variables are adjusted according to the properties stated above: the value of the auxiliary variable that stores the mapped clock value for clock x in state t is set to the maximum of the clock value of the x-zone representative of state t and the value of the auxiliary variable in state s. In an initial state, the auxiliary variable for clock x is initialized with the initial value I^x of the clock x. The actual definition of the refinement mapping from the extended real-time specification to the time-optimized specification then becomes

very simple: The refinement mapping maps a state s from the real-time system to a state that equals the state s except that the value of each clock $x \in \mathcal{X}$ is set to the value of the corresponding auxiliary variable for clock x in state s.

Remark. It follows directly from the definitions above that the values of the auxiliary variables do not change during $InnerNext_S$ steps. Therefore, steps of $InnerNext_S$ in the real-time system are mapped to steps in the time-optimized system in which the clock values remain constant.

Remark. It is an invariant of the extended real-time specification that the value for the auxiliary variable for clock x is always lower than or equal to the value of clock x. This can be seen based on an inductive argument: In an initial state, the value of the auxiliary variable for clock x equals the value of clock x. In each step in which the value of the auxiliary variable for clock x is changed, the new value of the auxiliary variable for clock x equals either (case A) the previous value or (case B) the clock value of a zone representative. Case A: The previous value of the auxiliary variable is always lower than or equal to the previous value of the clock x by the induction assumption and the previous value of clock x is always lower than or equal to the new value of clock x because, in a real-time specification, the value of a clock can only increase. Case B: The clock value of a state's zone representative is, by definition of a zone representative, always lower than or equal to the state's clock value. It follows that the auxiliary variable for clock x is always lower than or equal to the value of clock x.

The refinement mapping from the real-time system to the time-optimized system does not necessarily map a state to a zone representative. Instead, it maps a state s of the real-time system to a state $\hat{s}$ of the time-optimized system. We now derive a property that a zone must have to ensure that the state $\hat{s}$ simulates the state s so that the time-optimized system is an abstraction of the real-time system. The two states s and $\hat{s}$ might differ in the values of multiple clocks. The mapping from state s to state $\hat{s}$ can be divided into a sequence of states $s, s_1, s_2, \ldots, \hat{s}$ so that between two subsequent states only the value of one clock is changed. Our goal is that each state in this sequence of states is simulated by the following state, i.e. $s \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \ldots \rightsquigarrow \hat{s}$. From one state in this sequence of states to the following, say from state s_1 to state s_2, a single clock is updated. We call this clock x. The state s_2 is in the same x-zone as the state s_1 because the clock value of clock x in state s_2 is between the clock value of clock x of state s_1 and the clock value of clock x of the x-zone representative of state s_1. If a zone had the property that each state s in an x-zone is simulated by a state in the same zone where the clock x has the value $s.x - 1$, then it holds that the state s_1 simulates the state s_2. Therefore, we add this property to the definition of a zone and obtain the final definition of a zone as follows.

The definition of a zone contains five conditions that a subset of a real-time system's state space must fulfill to be a zone. Before the formal definition, we briefly explain these conditions informally: Condition 1 requires that the clocks of all states in a zone are equal except for the value of a specific clock $x \in \mathcal{X}$. The second condition requires that there are no 'gaps' in the zone. For example, if there are two states in the zone with the clock values 3 and 5 for the clock x, then the corresponding state with the clock value 4 must

also be in the zone. Condition 3 implies that every state in a zone is simulated by a state of the zone that has a lower clock value for the clock x. The last condition defines that a zone cannot be a subset of a larger zone which implies that a zone includes as many states as possible. The state with the lowest clock value for clock x is defined to be the zone representative.

Definition 9 (Zone, Zone Representative)**.** We define a *zone* to be a subset Z of a real-time system's state space Σ_S so that there exists a clock $x \in \mathcal{X}$ so that

1. in all states in the set Z, the values of all clocks $y \in \mathcal{X}$ so that $y \neq x$ are equal, and
2. for every two states z_1 and z_2 of the set Z it holds that if $z_1.x - z_2.x \geq 2$ then there is also a state z_3 in the set Z so that $z_1.x > z_3.x > z_2.x$, and
3. for every two states z_1 and z_2 of the set Z it holds that if $z_1.x = z_2.x - 1$ then the state z_1 simulates the state z_2 ($z_2 \rightsquigarrow z_1$), and
4. there is no subset of the state space Σ_S that fulfills the above conditions and is a proper superset of the set Z.

The state $z_1 \in Z$ of the zone for which it holds for all other states $z_2 \in Z$ of the zone that $z_1.x \leq z_2.x$ is called a *zone representative*. To explicitly name the clock x, a zone is also called an x-zone. The zone representative of an x-zone is an x-zone representative.

Remark. All states in a zone are simulated by the zone representative but, on the contrary, the zone representative is not necessarily simulated by other states in the zone.

Remark. By Definition 9, all states in a zone are simulated by the zone representative and, thus, the states in a zone are varequal to each other, i.e., they differ only by the values of their clocks.

Example 4.5. An example of a zone in the exemplary specification in Fig. 4.6 on page 96, is the set of all states that are varequal to the initial state (i.e., these states have the same values for the variable HTLCs) and whose value of the variable *Time* is in the set $\{1, ..., 9\}$. The initial state is the zone representative of this zone. This set fulfills the definition of a zone (Definition 9) for the following reasons: All of these states are varequal (condition 1 of Definition 9), the set contains states with all clock values between 1 and 9 (condition 2), and each state in the zone is simulated by a state in the zone that has a lower clock value (condition 3). Further, this set is the largest subset of the state space that fulfills the properties of a zone because, by condition 2 of Definition 9, a larger subset of the state space must contain the state where the clock has the value 10. Such a state with clock value 10 is not simulated by a state with clock value 9 because when the clock equals 10, one HTLC can be timed out which is not possible at clock value 9 (condition 4).

As an example of a time-optimized specification $\hat{S}$ whose state space contains only a single state for each zone, a time-optimized specification for the specification shown in Fig. 4.6 is given in Fig. 4.8. The specification differs from Fig. 4.6 only in the definition of the action *AdvanceTime*. Instead of advancing the variable *Time* to any future point in time, only the timelocks of all HTLCs that are in state NEW are possible new values.

module *SimpleHTLCsOptimized*

extends *Integers*

variables *Time*, *HTLCs*

$TypeOK \triangleq$
$\quad \land \quad Time \in Nat$
$\quad \land \quad HTLCs \subseteq [id : Nat,\ timelock : Nat,\ state : \{\text{“NEW”}, \text{“FULFILLED”}, \text{“TIMED OUT”}\}]$

$Init \triangleq$
$\quad \land\ Time = 1$
$\quad \land\ HTLCs = \{[id \mapsto 1,\ timelock \mapsto 10,\ state \mapsto \text{“NEW”}],$
$\qquad\qquad [id \mapsto 2,\ timelock \mapsto 20,\ state \mapsto \text{“NEW”}]\}$

ZoneTimePoints $\triangleq$
$\quad$ let newHTLCs $\triangleq$ { htlc $\in$ HTLCs : htlc . state = “NEW” }
$\quad$ in { htlc . timelock : htlc $\in$ newHTLCs }

$AdvanceTime \triangleq$
$\quad \land \exists\, n \in$ ZoneTimePoints $: Time' = n$
$\quad \land\ Time' > Time$
$\quad \land$ unchanged $HTLCs$

$Fulfill(htlc) \triangleq$
$\quad \land\ htlc.state = \text{“NEW”}$
$\quad \land\ Time < htlc.timelock$
$\quad \land\ HTLCs' = (HTLCs \setminus \{htlc\}) \cup \{[htlc \text{ except } !.state = \text{“FULFILLED”}]\}$
$\quad \land$ unchanged $Time$

$Timeout(htlc) \triangleq$
$\quad \land \quad htlc.state = \text{“NEW”}$
$\quad \land \quad Time \geq htlc.timelock$
$\quad \land \quad HTLCs' = (HTLCs \setminus \{htlc\}) \cup \{[htlc \text{ except } !.state = \text{“TIMED OUT”}]\}$
$\quad \land \quad$ unchanged $Time$

$Next \triangleq$
$\quad \lor\ AdvanceTime$
$\quad \lor \exists\, htlc \in HTLCs : Timeout(htlc) \lor Fulfill(htlc)$

$Spec \triangleq\ Init \land \Box[Next]_{\langle Time,\, HTLCs \rangle}$

Figure 4.8.: Time-optimized specification for the specification in Fig. 4.6. The changes to the specification in Fig. 4.6 are highlighted.

While similar definitions to our definition of simulation have already appeared in previous work [LV96; DT98; Bou+18], to the best of our knowledge, there is no proof that shows how to construct a time-optimized specification given a real-time specification as we specify it in Section 4.1.1. We fill this gap by defining a time-optimized specification and a refinement mapping from the real-time specification to the time-optimized specification and by proving that the real-time specification implements the time-optimized specification. This proof applies to all real-time specifications that follow the definition in Section 4.1.1 and, thus, generalizes beyond our TLA$^+$ specification of Lightning.

In this subsection, we have implicitly constructed a time-optimized specification by defining a refinement mapping from a real-time specification to a time-optimized specification. In the following subsection, we explicitly define this time-optimized specification.

4.1.3. Time-Optimized Specifications

Based on a real-time system S, we define the time-optimized system $\hat{S}$ with the time-optimized specification $Spec_{\hat{S}}$. This specification is defined as $Spec_S$ with the only difference how time is advanced by the $AdvanceTime$ action. Formally, $Spec_{\hat{S}}$ has the variables $v_{\hat{S}} = v_S$ and is defined as $Spec_{\hat{S}} = Init_{\hat{S}} \wedge \Box[Next_{\hat{S}}]_{v_{\hat{S}}} \wedge Liveness_{\hat{S}}$ with $Init_{\hat{S}} = Init_S$ and $Next_{\hat{S}} = AdvanceTime_{\hat{S}} \vee InnerNext_{\hat{S}}$ and $InnerNext_{\hat{S}} = InnerNext_S$ and $Liveness_{\hat{S}} = Liveness_S$. See Fig. 4.9 for an overview of the definitions. As above for specification $Spec_S$, we use $\hat{\Sigma}$ to refer to the state space of all possible states using the variables $v_{\hat{S}}$. To define $AdvanceTime_{\hat{S}}$, we introduce the following definitions.

We define a function T_d^x (T stands for time translation) that translates a state s_1 from the state space Σ of the real-time specification to another time by returning a state s_2 as a copy of state s_1 with the clock x set to d, i.e. $s_2.x = d$. Intuitively, T_d^x changes the value of clock x in state s_1 to the value d.

Definition 10 (Time translation of states, T_d^x)**.**

$$
\begin{aligned}
&T_d^x : \Sigma \to \Sigma \\
&\quad s_1 \mapsto s_2 \text{ so that } (s_2 \approxeq s_1) \\
&\qquad\qquad \wedge \forall y \in \mathcal{X} : s_2.y = \begin{cases} d & \text{if } y = x \\ s_1.y & \text{else} \end{cases}
\end{aligned}
$$

Example 4.6. In our sketches of an exemplary real-time specification, the application of T_d^x on a state is represented by moving a state horizontally on the axis for clock x to value d (see Fig. 4.10). Consider a state s with $s.x = 3$, i.e. the clock x has the value 3. The state $T_5^x(s)$ is a state that is varequal to state s but the value of clock x is set to 5, i.e. $T_5^x(s).x = 5$. The state $T_{s.x-1}^x(s)$ is the state that is varequal to state s but the value of the clock x is one time unit lower than in state s, i.e. $T_{s.x-1}^x(s).x = s.x - 1 = 2$.

$Init \triangleq$ initially,
- $\land \forall x \in X :$ all clocks
  - $\lor x = I[x]$ are set to their initial value or
  - $\lor x = INACTIVE$ are set to INACTIVE

$AdvanceTime \triangleq$
- $\land$ UNCHANGED v all non-clock variables are unchanged
- $\land \exists x \in X : x' > x$ at least one clock is advanced
- $\land \forall x \in X :$ for all clocks x in X
  - IF $x = INACTIVE$ the next value of clock x is
    - THEN $x' = INACTIVE$ INACTIVE, if the previous value of clock x is INACTIVE
    - ELSE $\lor$ UNCHANGED x else: either clock x is unchanged
      - $\lor \land x' > x$ or the value of clock x increases and
        - $\land x' \in relZTP(x)$ the new value of is in relZTP(x)
- $\land \forall x \in X :$ for all clocks x in X
  - $x \neq INACTIVE \implies$ if the clock is active
    - $\forall b \in TimeBounds(x) : x \leq b \implies x' \leq b$ no time bound is skipped

$InnerNext \triangleq$
- $\land \ldots$
- $\land \forall x \in X :$
  - $\land$ IF $x = INACTIVE$
    - THEN $x' = I[x] \lor$ UNCHANGED x inactive clocks are activated or unchanged
    - ELSE UNCHANGED x active clocks are unchanged

$Next \triangleq InnerNext \lor AdvanceTime$

$Spec \triangleq Init \land \Box[Next]_{(v \cup X)} \land Liveness$

Figure 4.9.: Pseudo-TLA$^+$ specification of a time-optimized system to give an overview of the definitions in Section 4.1.3. The specification does not contain the subscripts for the system $\hat{S}$ ($_{\hat{S}}$) because these are not allowed in TLA$^+$ syntax. The specification differs to Fig. 4.4 only in the definition of the $AdvanceTime$ action.

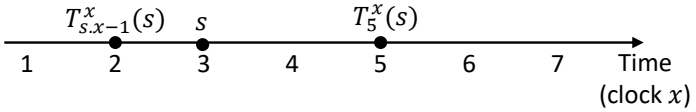

Figure 4.10.: Example of the application of the function T^x_d on a state s. In state s, the value of clock x is 3. The state $T^x_5(s)$ is a state that is varequal to state s but the value of the clock x is set to 5. The state $T^x_{s.x-1}(s)$ is a state that is varequal to state s but the value of the clock x is 2 which is one unit of time lower than the value of the clock x in state s.

The idea of the time-optimized specification is that the action $AdvanceTime_{\hat{S}}$ advances time only to the clock values of a zone representative. As shown in Section 4.1.2, this implies that each state s of the real-time system S is mapped to a state that simulates s. We refer to the points in time that are clock values of a zone representative as zone time points.

Example 4.7. A set of zone time points for the example specification in Fig. 4.6 on page 96 is the set that contains the timelocks of all HTLCs that are in state NEW. In the initial state, the set of zone time points equals $\{10, 20\}$ because both HTLCs are in state NEW.

To determine the points in time to which a clock $x \in \mathcal{X}$ can advance in the action $AdvanceTime_{\hat{S}}$ of the time-optimized specification, we need the clock values of clock x of zone representatives, i.e. the zone time points for clock x. In the following, we derive a definition of these zone time points. We could define the set of zone time points for clock x in a first attempt informally as the set of natural numbers d for which a state t exists that is a zone representative and the value of the clock x in state t equals the number d:

$$\{d \in \mathbb{N} \mid \exists t \in \Sigma : t.x = d \wedge t \text{ is a zone representative}\}$$

As an optimization, we do not need to consider the complete set of zone representatives but it suffices to consider zone representatives that are reachable from the current state. Thus, we define the set of zone time points dependent on state s. We refine the first attempt to the following definition that considers only zone representatives that are reachable from state s. We write $s \xrightarrow{A^+} t$ if state t can be reached from state s by a sequence of one or more A steps. Thus, $s \xrightarrow{([Next_S]_{v_S})^+} t$ means that the state t can be reached from state s by a sequence of one or more $Next_S$ or stuttering steps.

$$\begin{aligned}\{d \in \mathbb{N} \mid \exists t \in \Sigma : s &\xrightarrow{([Next_S]_{v_S})^+} t \\ &\wedge\ t.x = d \\ &\wedge\ t \text{ is a zone representative} \\ \}&\end{aligned}$$

In a next step, we formalize the condition that the state t is a zone representative. Consider a state t that is in a zone Z that contains states whose clock values of the clock x differ. The equivalence that follows in Equation (4.1) is proven in the appendix in Section A.3.1. Equation (4.1) shows that the state t is a zone representative if and only if the state t is not simulated by the state $T^x_{t.x-1}(t)$ which is the state t with the clock value of the clock x reduced by 1:

$$t \text{ is a zone representative} \Leftrightarrow \neg(t \rightsquigarrow T^x_{t.x-1}(t)) \tag{4.1}$$

Consequently, we could define the set of zone time points for state s as follows:

$$\begin{aligned}\{d \in \mathbb{N} \mid \exists t \in \Sigma : s \xrightarrow{([Next_S]_{v_S})^+} t \\ \wedge\ t.x = d \\ \wedge\ \neg(t \rightsquigarrow T^x_{t.x-1}(t)) \\ \}\end{aligned} \tag{4.2}$$

The formalization of the set of zone time points in Eq. (4.2) makes it difficult to constructively find the clock values of zone representatives because it needs to be checked whether a state simulates another state. Therefore, we formally define the function ZTP^x in Definition 11 by replacing the last condition. Compared to Eq. (4.2), Definition 11 replaces the expression $t \rightsquigarrow T^x_{t.x-1}(t)$ by an expression that considers only a single step which facilitates proving that a certain point in time is contained in the set $ZTP^x(s)$ for a state s and a clock $x \in \mathcal{X}$. We prove in the appendix in Section A.3.2 that the following definition is equivalent to the definition in Eq. (4.2). After the formal definition, we explain the definition with an example shown in Fig. 4.11.

Definition 11 (Zone time points of clock x for states, ZTP^x)**.** The function ZTP^x (ZTP stands for Zone Time Points) maps for each clock $x \in \mathcal{X}$ a state s to the zone time points of the zone representatives that are reachable from state s.

$$\begin{aligned}&ZTP^x : \Sigma \to \mathcal{P}(\mathbb{N}) \\ &\qquad s \mapsto \{d \in \mathbb{N} \mid \exists t \in \Sigma : s \xrightarrow{([Next_S]_{v_S})^+} t \\ &\qquad\qquad \wedge\ t.x = d \\ &\qquad\qquad \wedge\ \exists u \in \Sigma : t \xrightarrow{Next_S} u \wedge \neg(T^x_{d-1}(t) \xrightarrow{Next_S} T^x_{u.x-1}(u)) \\ &\qquad \}\end{aligned}$$

The definition of ZTP^x can be read as follows. The function ZTP^x is defined as a set of time points d that satisfy the following condition: Given a state s, there exists a state t that is reachable from state s by a series of $Next_S$ steps. The example in Fig. 4.11 shows a state t that can be reached from state s by a series of $AdvanceTime_S$ steps (horizontal arrows) and $InnerNext_S$ steps (vertical arrows). The state t is a zone representative if it holds that there exists a state u that is reachable from state t by a $Next_S$ step and, stated informally, the same step is not possible at the directly preceding point in time. The condition that the same step is not possible at the directly preceding point in time is formally expressed using the notation $T^x_{t.x-1}(t)$ for the state which is obtained by reducing the value of the clock x of state t by one unit of time: Given that the step from state $T^x_{t.x-1}(t)$ to state $T^x_{u.x-1}(u)$ (state u but the value of clock x reduced by one unit of time) is not a $Next_S$ step, the state t is not simulated by the state $T^x_{t.x-1}(t)$ and, thus, state t must be a zone representative.

While the function ZTP^x defines the zone time points as they are required for the time-optimized specification, it might be difficult to explicitly define the function ZTP^x for a

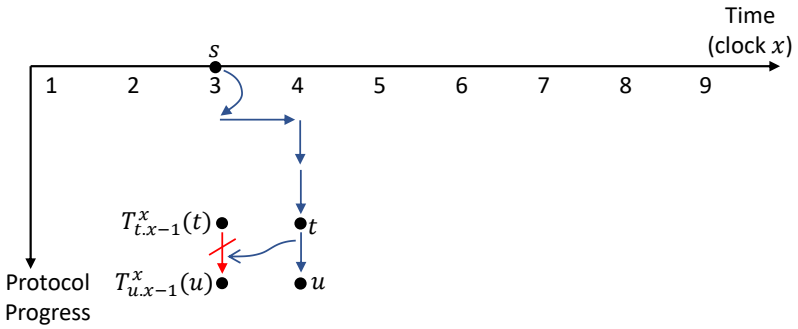

Figure 4.11.: Example of zone time points ($ZTP^x(s)$, see Definition 11). For simplicity, the example shows a single clock. The state s has the clock value 3. There exists a state t so that the state t can be reached from state s by a series of $AdvanceTime_S$ and $Next_S$ steps. $AdvanceTime_S$ steps are represented by horizontal blue arrows and $InnerNext_S$ steps are represented by vertical blue arrows. There exists a state u that is reachable from state t by a $Next_S$ step. The state $T^x_{t.x-1}(t)$ is defined as the state that equals state t except that the value of the clock x in state t is reduced by 1. The state $T^x_{u.x-1}(u)$ is defined analogously for state u. In this scenario, the step from state $T^x_{t.x-1}(t)$ to state $T^x_{u.x-1}(u)$ is not a $Next_S$ step. By definition of $ZTP^x(s)$, the clock value of state t (4) is a zone time point.

specific specification. To facilitate an explicit definition in specification $Spec_{\hat{S}}$, we require that, instead of explicitly defining ZTP^x, for each clock $x \in \mathcal{X}$, a function $relZTP^x(s)$ is defined that maps a state to a set of natural numbers so that $relZTP^x(s)$ is a superset of $ZTP^x(s)$. Defining $relZTP^x$ is easier than defining ZTP^x because it needs to be ensured only that all points in time that are in the set $ZTP^x(s)$ are in the set $relZTP^x(s)$ but not that the set $relZTP^x(s)$ equals exactly the set $ZTP^x(s)$. However, for the largest reduction of the state space from the original specification to the time-optimized specification, a set $relZTP^x(s)$ should contain as few points in time as possible.

Example 4.8. Consider the exemplary system in Fig. 4.8 (on page 105). The definition of *ZoneTimePoints* could be replaced by the definition $ZoneTimePointsLarge \triangleq \{htlc.timelock : htlc \in HTLCs\}$ which is a superset of ZTP^{Time} because the timelock of an HTLC is included even if the HTLC has already been fulfilled or timed out. While in this simple example, the definition of $ZoneTimePointsLarge$ is only slightly simpler than the definition of *ZoneTimePoints*, such simplifications facilitate more complex specifications.

To be able to use the set $relZTP^x(s)$ instead of the set $ZTP^x(s)$, the definition of $relZTP^x$ needs to fulfill the following assumptions that ensure that some properties of the function ZTP^x also hold for the definition of $relZTP^x$. The first assumption on $relZTP^x$ is, as we have already stated above, that $relZTP^x(s)$ is a superset of $ZTP^x(s)$:

Assumption 4.2. *For any state $s \in \Sigma$ and all clocks $x \in \mathcal{X}$, it holds that*

$$ZTP^x(s) \subseteq relZTP^x(s)$$

Remark. Because $relZTP^x(s)$ is a superset of $ZTP^x(s)$, each state of the real-time specification is still simulated by a state of the time-optimized spec if the set $ZTP^x(s)$ is replaced by the set $relZTP^x(s)$.

For the set $ZTP^x(s)$ it holds for each step from state s to state t that the set $ZTP^x(t)$ is a subset of the set $ZTP^x(s)$, i.e. that the step can only reduce the set of zone time points but no new zone time points can appear after a step. The reason for this is that all zone representatives that are reachable from state t are also reachable from state s. However, the two sets $ZTP^x(s)$ and $ZTP^x(t)$ can be different if zone representatives that were reachable from state s are not reachable from state t anymore.

With the following assumption, we require that the definition of the set $relZTP^x(s)$ also has the property that after a step of $Next_S$ from state s to state t the set $relZTP^x(t)$ is smaller than or equal to the set $relZTP^x(s)$, i.e., there is no value in the set $relZTP^x(t)$ that was not contained in the set $relZTP^x(s)$ before the step.

Assumption 4.3. *For any two states $s, t \in \Sigma$ and all clocks $x \in \mathcal{X}$, it holds that, if the step $s \longrightarrow t$ is a $Next_S$ step, then*

$$relZTP^x(t) \subseteq relZTP^x(s)$$

Based on Assumptions 4.2 and 4.3, we define the mapping $relZTP^x$ as follows.

Definition 12 ($relZTP^x$, relaxed Zone Time Points)**.** Define $relZTP^x$ for a clock $x \in \mathcal{X}$ to be a mapping from the state space Σ of the real-time specification S to a set of natural numbers so that Assumptions 4.2 and 4.3 hold.

Using the definition of $relZTP^x$, we define the action $AdvanceTime_{\hat{S}}$ as an action that sets the new value of each clock $x \in \mathcal{X}$ to a point in time that is in the set $relZTP^x$. All other variables are left unchanged:

Definition 13 ($AdvanceTime_{\hat{S}}$)**.** For two states $s, t \in \hat{\Sigma}$, it holds that $s \xrightarrow{AdvanceTime_{\hat{S}}} t$ if and only if

$$\begin{aligned}
& t \approxeq s \\
& \quad \wedge \exists x \in \mathcal{X} : t.x > s.x \\
& \quad \wedge \forall x \in \mathcal{X} : \begin{cases} t.x = \bot & \text{if } s.x = \bot \\ t.x = s.x \vee (t.x \in relZTP^x(s) \wedge t.x \geq s.x) & \text{else} \end{cases} \\
& \quad \wedge \forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b
\end{aligned}$$

Described informally, this definition means that a step from state s to state t is an $AdvanceTime_{\hat{S}}$ step if and only if

- the states s and t are varequal ($t \approxeq s$),
- there exists a clock x whose value increases in the step ($t.x > s.x$),

- for all active clocks x it holds that either the clock value remains unchanged ($t.x = s.x$) or the clock value increases ($t.x \geq s.x$) to a zone time point ($t.x \in relZTP^x(s)$), and inactive clocks remain unchanged, and
- for all active clocks x it holds that no time bound is skipped during the step, i.e. if there is a time bound b so that $s.$ was below the time bound ($s.x \leq b$), then $t.x$ must still be below the time bound ($t.x \leq b$).

With the definition of $AdvanceTime_{\hat{S}}$, the specification $Spec_{\hat{S}}$ is fully defined and we can state the theorem that $Spec_S$ implements $Spec_{\hat{S}}$:

Theorem 1.

$$Spec_S \Rightarrow Spec_{\hat{S}}$$

We prove Theorem 1 in Section 4.1.5. Before that, we introduce an extension of the real-time specification that adds auxiliary variables required for the proof. This extension is used to define the mapping from the real-time specification to the time-optimized specification.

4.1.4. Extended Real-Time Specifications

The idea of the proof that the real-time specification $Spec_S$ implements the time-optimized specification $Spec_{\hat{S}}$ is to define a mapping that maps states of the real-time specification $Spec_S$ to states of the time-optimized specification $Spec_{\hat{S}}$. As explained in Section 4.1.2, the definition of a refinement mapping from the real-time specification to the time-optimized specification needs auxiliary variables. These auxiliary variables are added to the real-time specification in the extended real-time specification that we define in the following. The auxiliary variables store for each clock $x \in \mathcal{X}$ the value to which the clock is mapped in the time-optimized specification.

Specification $Spec_{S'}$ uses the variables $v_{S'}$ defined as the variables v_S of $Spec_S$ and the auxiliary variables $mappedClock^x$ for each clock $x \in \mathcal{X}$:

$$v_{S'} = v_S \cup \bigcup_{x \in \mathcal{X}} \{mappedClock^x\}$$

Specification $Spec_{S'}$ is defined by $Spec_{S'} = Init_{S'} \wedge \Box[Next_{S'}]_{v_{S'}} \wedge Liveness_{S'}$ where $Init_{S'}$ is defined as follows. All variables and clocks that exist in the real-time specification $Spec_S$ are initialized as in $Spec_S$ and the auxiliary variables for each clock are initialized with the corresponding clock's initial value.

Definition 14 ($Init_{S'}$)**.**

$$Init_{S'} = Init_S \wedge \forall x \in \mathcal{X} : mappedClock^x = x$$

$Liveness_{S'}$ of the extended real-time specification is defined as $Liveness_S$ of the original real-time specification.

$Next_{S'}$ is defined as $Next_{S'} = AdvanceTime_{S'} \vee InnerNext_{S'}$ where $InnerNext_{S'}$ is defined based on $InnerNext_S$ and a condition for the new values of the $mappedClock^x$ variables. The value of the variable $mappedClock^x$ remains unchanged if the respective clock x is unchanged during the step $InnerNext_S$. The only case in which the value of a clock x can change is when a clock is activated and the clock's value is changed from $\perp$ to the clock's initial value I^x. In this case, the value of $mappedClock^x$ is set to the new value of the clock variable. Formally:

Definition 15 ($InnerNext_{S'}$)**.** A step from state s to state t is an $InnerNext_{S'}$ step if and only if the step from state s to state t is an $InnerNext_S$ step and it holds that

$$\forall x \in \mathcal{X} : t.mappedClock^x = \begin{cases} s.mappedClock^x & \text{if } t.x = s.x \\ t.x & \text{else} \end{cases}$$

Given a state $s \in \Sigma'$, we use the notation $\ddot{s}$ to refer to a copy of state s of which the variables $mappedClock^x$ were removed so that the state $\ddot{s}$ is in the state space Σ of the original real-time specification. As this notation is only needed for formal correctness, for an intuitive understanding, the state $\ddot{s}$ can be read as the state s.

$AdvanceTime_{S'}$ is defined as follows and informally explained after the definition:

Definition 16 ($AdvanceTime_{S'}$)**.** For two states $s, t \in \Sigma'$, it holds that $s \xrightarrow{AdvanceTime_{S'}} t$ if and only if

$$(\ddot{s} \xrightarrow{AdvanceTime_S} \ddot{t})$$
$$\wedge \forall x \in \mathcal{X} : t.mappedClock^x = \begin{cases} \perp & \text{if } t.x = \perp \\ max(\quad \{s.mappedClock^x\} \cup & \\ \qquad \{n \in relZTP^x(\ddot{s}) \mid n \leq t.x\}) & \text{else} \end{cases}$$

Informally, this definition means that a step from state s to state t is an $AdvanceTime_{S'}$ step if and only if the change of all variables except the $mappedClock^x$ variables is described by the action $AdvanceTime_S$ of the original real-time specification and the values of the $mappedClock^x$ variables in state t are set as follows. If the corresponding clock x is inactive, the value of $mappedClock^x$ is set to $\perp$. Otherwise, the main idea is to set the value of $mappedClock^x$ to the clock value of the zone representative of the zone that contains the state t (see Fig. 4.12). Because the definition of $AdvanceTime_{\ddot{S}}$ uses the set $relZTP^x$ which might contain more values than the set ZTP^x, the value of the mapped clock might be larger than the clock value of the zone representative but the condition also ensures that the value of the mapped clock is not larger than the respective clock value $t.x$. If the

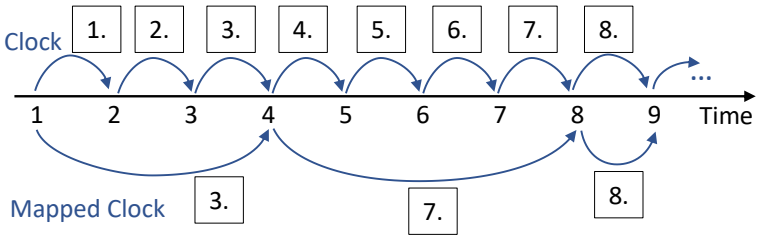

Figure 4.12.: Time flow of an exemplary extended real-time specification that has an additional mapped clock variable for each clock. The blue arrows indicate steps of the action *AdvanceTime*. For simplicity, the example uses only a single clock. As in Fig. 4.2, the exemplary specification has tree zones starting at time units 1, 4, and 7. The numbers in the boxes indicate the order in which steps are taken: In the first two steps, the value of the clock is advanced but the value of the mapped clock does not change. When the value of the clock advances to 4 in the third step, the value of the mapped clock advances to 4, too. Similarly, the fourth and fifth step change only the clock's value while the sixth step advances the values of the clock and the mapped clock to 7.

value of the mapped clock is larger than the clock value of the zone representative because the definition of $relZTP^x$ adds additional time points, the correctness of the approach is unconcerned but the optimization is not optimal. To ensure that the value of the mapped clock never decreases during a step of $AdvanceTime_{S'}$, a further condition ensures that the value of the mapped clock in state t is set to at least the mapped clock's value in state s.

The refinement mapping from the extended real-time system S' to the time-optimized system $\hat{S}$ will be defined by setting each clock x in the time-optimized system $\hat{S}$ to the value of the corresponding variable $mappedClock^x$.

4.1.5. Proof of Theorem 1

The general idea to prove Theorem 1 is to show that the original real-time system S implements the extended system S' (see Lemma 1) and that the extended system S' implements the time-optimized system $\hat{S}$ (see Lemma 3). Figure 4.3 on page 91 gives an overview of this approach: We first prove that the original system S implements the extended system S' in which the $mappedClock$ variables are added (see Lemma 1). Then we prove that the extended system S' implements the time-optimized system $\hat{S}$ by showing that each behavior of the extended system S' can be mapped to a behavior of the time-optimized system $\hat{S}$ using the mapping Λ that sets the values of the clocks in system $\hat{S}$ to the values of the corresponding $mappedClock$ variables of system S' (see Lemma 3).

The proof of Lemma 1 is trivial because the variables that are added in $Spec_{S'}$ compared to $Spec_S$ do not affect the behaviors allowed by the specification $Spec_{S'}$ and, thus, all behaviors allowed by the original specification $Spec_S$ are also allowed by the extended specification $Spec_{S'}$. To prove Lemma 3, we first prove that an invariant $Inv_{S'}$ holds for $Spec_{S'}$. The invariant $Inv_{S'}$ guarantees that by applying the mapping Λ which sets a clock x to the value of the corresponding $mappedClock^x$ variable, a state is mapped to a state

that is in the same zone. Then, we use the invariant $Inv_{S'}$ to prove that $Spec_{S'}$ implements $Spec_{\hat{S}}$.

We introduce the following notation: $\Lambda(F)$ is formula F in which each clock $x \in \mathcal{X}$ is replaced by the variable $mappedClock^x$. Formally, this can be expressed in TLA$^+$ using the notation F WITH $v_1 \leftarrow e_1, v_2 \leftarrow e_2$ which equals the expression F in which variable v_1 is substituted by expression e_1 and variable v_2 is substituted by expression e_2. The formal definition of $\Lambda(F)$ is as follows.

Definition 17 (Clock replacement, $\Lambda(F)$)**.** Let $x_1, x_2, x_3, ...$ be the clocks in the set of clocks $\mathcal{X}$. For a formula F, the value of $\Lambda(F)$ is the formula F in which every occurrence of the clock variable x_i is replaced by the corresponding variable $mappedClock^{x_i}$ for every $i \in \{1, \ldots, |\mathcal{X}|\}$. Formally, using TLA$^+$ syntax:

$$\begin{aligned}\Lambda(F) = F \text{ WITH } x_1 &\leftarrow mappedClock^{x_1},\\ x_2 &\leftarrow mappedClock^{x_2},\\ x_3 &\leftarrow mappedClock^{x_3},\\ &\ldots\end{aligned}$$

The following lemma shows that the original real-time specification implements the extended real-time specification.

Lemma 1. $Spec_S \Rightarrow \exists\!\!\exists\, mappedClock^{x_1}, ..., mappedClock^{x_{|\mathcal{X}|}} : Spec_{S'}$

The operator $\exists\!\!\exists\, v : F$ is a temporal existential quantifier that can be read simply as 'there exists a variable v so that F holds'. For an intuitive understanding, the lemma can be read as $Spec_S \Rightarrow Spec_{S'}$. However, formally, the $\exists\!\!\exists$ operator is required because the $mappedClock^x$ variables appear only in the formula on the right side of the implication.

Proof: Lemma 1 follows directly from the definition of $Spec_{S'}$ because the added variables $mappedClock^x$ are auxiliary variables that are not used in conditions but are only updated during steps of $Spec_{S'}$. Thus, these variables do not affect whether a $Next_S$ step is enabled or not (see [LM17, Theorem 1]). □

To prove Lemma 3, we first define the invariant $Inv_{S'}$ which asserts that the mapped value $mappedClock^x$ of a clock x must be larger than or equal to the largest zone time point that is lower than the value of the clock x. This invariant implies that each clock of a state s is mapped to a value so that the states s and $\Lambda(s)$ are in the same zone.

Definition 18 ($Inv_{S'}$)**.** Define an invariant $Inv_{S'}$ of a state s of specification $Spec_{S'}$ as:

$$Inv_{S'}(s) \equiv \forall x \in \mathcal{X} : s.mappedClock^x \geq max(\{n \in relZTP^x(\ddot{s}) \mid n \leq s.x\})$$

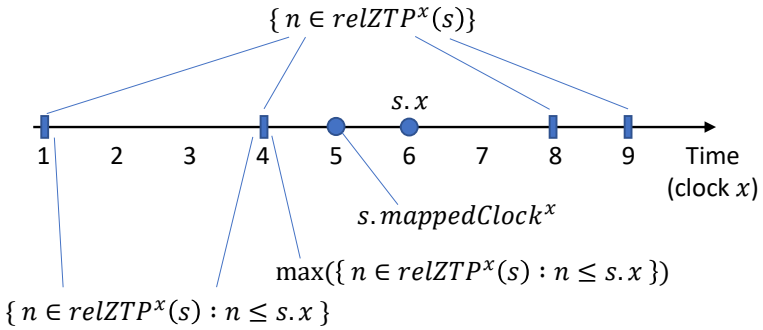

Figure 4.13.: Visualization of the invariant $Inv_{S'}$ based on an exemplary scenario. There are four zone time points shown by blue boxes (at 1, 4, 8, 9). Also the values of the clock x (6) and the variable $mappedClock^x$ (5) for a state s are shown. The terms of the invariants definition are visualized showing that the invariant is fulfilled in the shown scenario because $max(\{n \in relZTP^x(\breve{s}) \mid n \leq s.x\}) = 4 \leq 5 = s.mappedClock^x$.

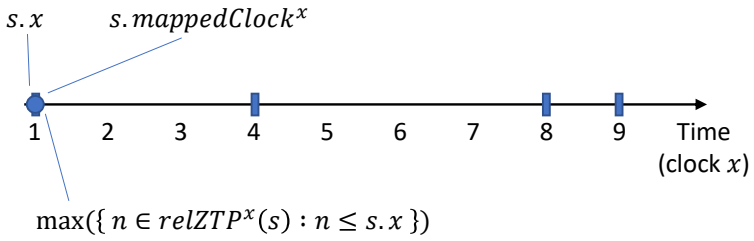

Figure 4.14.: Visualization of an exemplary scenario showing an initial state in which the value of clock x is set to 1. By definition of $Init_{S'}$ (see Definition 14 on page 112), $mappedClock^x$ is also initialized to 1. Because the largest zone time point that is lower than $s.x$ is also 1, the invariant is fulfilled.

Example 4.9. A visualization of an exemplary scenario is shown in Fig. 4.13 where the terms of Definition 18 are instantiated with specific values.

The following lemma shows that the invariant $Inv_{S'}$ is an invariant of specification $Spec_{S'}$.

Lemma 2. *From Assumption* 4.3*, it follows that $Inv_{S'}$ is an invariant of specification $Spec_{S'}$:*

$$Spec_{S'} \Rightarrow \Box Inv_{S'}$$

We prove Lemma 2 in the appendix in Section A.3.3. Here, we sketch the proof. To prove that the invariant $Inv_{S'}$ holds for the specification $Spec_{S'}$, we show that the invariant $Inv_{S'}$ is satisfied by the initial states (i.e., that $Init_{S'} \Rightarrow Inv_{S'}$) and then we show that the invariant $Inv_{S'}$ is maintained during $Next_{S'}$ steps, i.e., that, if the invariant holds in the starting state of a $Next_{S'}$ step, then the invariant holds in the ending state of the step which is formally expressed as $Inv_{S'} \wedge Next_{S'} \Rightarrow Inv_{S'}'$.

It follows from the definitions of $Init_{S'}$ and $Inv_{S'}$ that the invariant $Inv_{S'}$ holds for the initial states. An exemplary scenario for the invariant in an initial state is shown in Fig. 4.14.

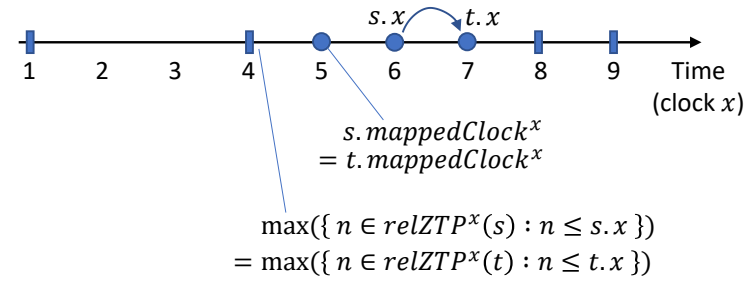

Figure 4.15.: Exemplary scenario showing that the invariant $Inv_{S'}$ is maintained during an $AdvanceTime_{S'}$ step from state s to state t that does not change the value of the variable $mappedClock^x$. This is the case if the new value of the clock x ($t.x$) is not a zone time point.

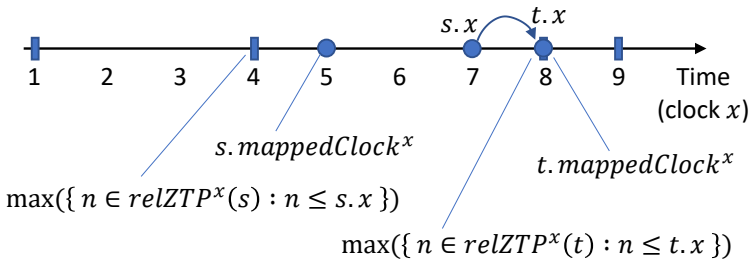

Figure 4.16.: Exemplary scenario showing that the invariant $Inv_{S'}$ is maintained during an $AdvanceTime_{S'}$ step from state s to state t that changes the value of the variable $mappedClock^x$ because the clock x advances to a zone time point.

To show that the invariant $Inv_{S'}$ is maintained during $Next_{S'}$ steps, we separately discuss $AdvanceTime_{S'}$ steps and $InnerNext_{S'}$ steps. Based on Assumption 4.3 which implies that the set $relZTP^x(s)$ does not increase during a step of $AdvanceTime_{S'}$, we show in the proof that the invariant $Inv_{S'}$ is maintained during an $AdvanceTime_{S'}$ step. Intuitively, this is either because the values of the $mappedClock^x$ variables do not change (see Fig. 4.15) or, if the value of a $mappedClock^x$ variable changes, the new value of $mappedClock^x$ is set to a zone time point (see Fig. 4.16).

For an $InnerNext_{S'}$ step, there are two cases that need to be considered for each clock: Typically, the clock's value is unchanged during the $InnerNext_{S'}$ step. However, in case that an inactive clock is activated, it can also be that a clock's value is changed. If a clock's value is unchanged then also the value of the mapped clock variable is unchanged. Because, by Assumption 4.3, the set $relZTP^x(s)$ does not increase during an $InnerNext_{S'}$ step, it follows that the invariant $Inv_{S'}$ is maintained during an $InnerNext_{S'}$ step. If a clock is activated during an $InnerNext_{S'}$ step, the invariant $Inv_{S'}$ is maintained because, in this case, the new value of the mapped clock equals the new value of the clock itself similarly to the scenario shown in Fig. 4.14. □

Based on Lemma 2, we prove that the extended specification $Spec_{S'}$ implements the time-optimized specification $Spec_{\hat{S}}$. This is formalized as the following lemma:

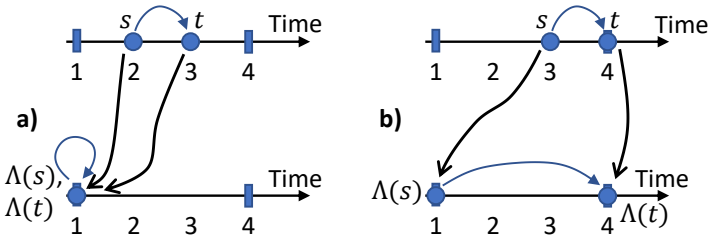

Figure 4.17.: Visualization of the mapping of an exemplary step in specification $Spec_{S'}$ (top) to specification $Spec_{\hat{S}}$ (bottom). For simplicity, we show only a single clock. The zone time points for the current state are indicated by blue boxes. The left part (a) of the figure shows an $AdvanceTime_{S'}$ step from state s to state t that does not change the value of the mapped clock because the clock's new value does not match a zone time point. Because the value of the mapped clock variable does not change in specification $Spec_{S'}$, the value of the clock in specification $Spec_{\hat{S}}$ does not change. Therefore, this step is mapped to a stuttering step in $Spec_{\hat{S}}$. The right part (b) of the figure shows an $AdvanceTime_{S'}$ step from state s to state t that changes the value of the mapped clock because the clock's value in state t equals a zone time point. This step is mapped to an $AdvanceTime_{\hat{S}}$ step in specification $Spec_{\hat{S}}$.

Lemma 3. *From* $Assumptions$ 4.1 *and* 4.2*, it follows that*

$$Spec_{S'} \Rightarrow \Lambda(Spec_{\hat{S}})$$

We prove Lemma 3 in the appendix in Section A.3.4 and sketch the proof ideas here. The mapping Λ maps states from the extended system S' to states of the time-optimized system $\hat{S}$ by setting the clocks' values in system $\hat{S}$ to the values of the respective mapped clocks in system S'. In the proof, we show that with this mapping all behaviors of the extended system S' are mapped to behaviors of the time-optimized system $\hat{S}$. We first show that the initial states of the extended system S' are mapped to initial states of the time-optimized system $\hat{S}$ and then we show that each step of the extended system S' is mapped to a step of the time-optimized system $\hat{S}$. Thereby, it follows that all behaviors of the extended system S' are mapped to behaviors of the time-optimized system $\hat{S}$.

It follows directly from the definitions of the $Init$ predicates in specification $Spec_{S'}$ and $Spec_{\hat{S}}$ that initial states of the extended system S' are mapped to initial states of the time-optimized system $\hat{S}$. We further show that steps of system S' are mapped to steps of system $\hat{S}$. More concretely, this proof idea is as follows: Consider, for example, the scenarios shown in Fig. 4.17 that we will discuss in the next paragraph. Figure 4.17 shows at the top a step from state s to state t in specification $Spec_{S'}$. These two states s and t are mapped to states $\Lambda(s)$ and $\Lambda(t)$ in specification $Spec_{\hat{S}}$ shown at the bottom of Fig. 4.17. The mapping Λ sets the value of each clock x in specification $Spec_{\hat{S}}$ to the value of the corresponding $mappedClock^x$ variable in specification $Spec_{S'}$. To prove that a step in specification $Spec_{S'}$ is mapped to a step in $Spec_{\hat{S}}$, we prove that the same action that describes the step from state s to state t describes the step from the mapped state $\Lambda(s)$ to the mapped state $\Lambda(t)$.

We show this by distinguishing the following three cases. The first case is that the $Next_{S'}$ step is a stuttering step or a step that changes only the values of the clocks but not the

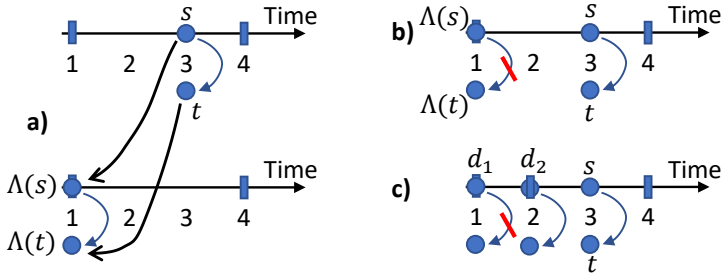

Figure 4.18.: Exemplary scenario of mapping a $Next_{S'}$ step from specification $Spec_{S'}$ (top) to specification $Spec_{\hat{S}}$ (bottom). In the left part (a), a $Next_{S'}$ step is shown that occurs at time 3 in specification $Spec_{S'}$. This step is mapped to time 1 in specification $Spec_{\hat{S}}$. The right part (b and c) sketches the idea of the contradiction proof. In this part, the states in the two specifications $Spec_{S'}$ and $Spec_{\hat{S}}$ are drawn on the same time axis. In part b), the same scenario as in part a) is shown: two states s and t at time 3 are mapped to two states at time 1. However, the figure shows the idea that we assume that the step between the mapped states is not allowed by specification $Spec_{\hat{S}}$. In the proof, we show that, in such a scenario, there are two consecutive points in time that are between the points in time 1 and 3 so that the step is not possible at the lower point in time but not at the greater point in time. This is sketched in part c where the lower point in time is 1 and the greater point in time is 2 at which the step is possible. This scenario implies that the time point 2 is a zone time point precisely because the step at time 2 is not possible at time 1. This is, however, a contradiction to the invariant $Inv_{S'}$.

mapped clock values (see Fig. 4.17a). In this case, the step is mapped to a stuttering step in $Spec_{\hat{S}}$.

The second case is that the $Next_{S'}$ step is an $AdvanceTime_{S'}$ step that changes the values of the mapped clocks (see Fig. 4.17b). We prove that, in this case, the step is mapped to an $AdvanceTime_{\hat{S}}$ step by showing that the definition of $AdvanceTime_{\hat{S}}$ (see Definition 13 on page 111) is fulfilled for a step of the mapped states. For example, we show that, if the value of a clock x changes, the new value is in the set $relZTP^x(s)$ as required by the definition of $AdvanceTime_{\hat{S}}$. A further requirement of the action $AdvanceTime_{\hat{S}}$ for the mapped step in $Spec_{\hat{S}}$ is that the step does not skip a time bound. To prove this statement, we use the assumption that the time bounds for a state s do not change if a clock value in state s is changed (see Assumption 4.1).

The third case is that the $Next_{S'}$ step is an $InnerNext_{S'}$ step. If all $mappedClock^x$ variables are equal to the corresponding clock values, the states s and t are mapped to the same states and it follows directly that the $InnerNext_{S'}$ step in specification $Spec_{S'}$ is mapped to an $InnerNext_{S'}$ step in specification $Spec_{\hat{S}}$. However, it might be that the states s and t are mapped to a different time value as shown in Fig. 4.18a. In this case, it must be proven that the step from the state $\Lambda(s)$ to the state $\Lambda(t)$ is an $InnerNext_{\hat{S}}$ step or, stated more informally, that the step can already be taken at the possibly lower time values of the mapped clocks. We prove this with a proof by contradiction based on the invariant $Inv_{S'}$: We assume that the step between the mapped states would not be a $Next_{\hat{S}}$ step (see Fig. 4.18b at time 1). This means that there exist two consecutive points in time d_1 and d_2 with $d_1 + 1 = d_2$ that are between the values $s.x$ and $s.mappedClock^x$ ($s.mappedClock^x \leq d_1 \wedge d_2 \leq s.x$) so that the step is not allowed at the lower point in time d_1 but that the step is allowed at the greater point in time d_2 (see time points $d_1 = 1$

and $d_2 = 2$ that show one possible scenario in Fig. 4.18c). By the definition of zone time points (see Definition 11), this implies that the greater point in time d_2 (time point 2 in Fig. 4.18c) is a zone time point. Because the time point d_2 is lower than $s.x$, the invariant $Inv_{S'}$ implies that the value of $s.mappedClock^x$ must be greater than or equal to the time point d_2. This is a contradiction because the time point d_2 is by definition greater than $s.mappedClock^x$. Therefore, the assumption that the step between the mapped states would not be a $Next_{\hat{S}}$ step must be wrong, i.e. the mapped step is a $Next_{\hat{S}}$ step.

Having shown that the initial states of the extended system S' are mapped to initial states of the time-optimized system $\hat{S}$ and that each step of the extended system S' is mapped to a step of the time-optimized system $\hat{S}$, it follows that all behaviors of the extended system S' are mapped to behaviors of the time-optimized system $\hat{S}$. □

With these lemmas, we can prove Theorem 1 which was first introduced on page 112 and is repeated here:

Theorem 1.

$$Spec_S \Rightarrow Spec_{\hat{S}}$$

Proof: Theorem 1 follows directly from Lemma 1 and Lemma 3. □

4.1.6. Conclusion

In this section, we defined a general real-time specification and a time-optimized specification that potentially has a smaller state space than the original real-time specification. We proved that the original real-time specification implements the time-optimized specification based on a few assumptions on the definition of the time-optimized specification (namely on the definition of the set of relaxed zone time points $relZTP^x$ and the set of time bounds B^x). We will later show that the specification of Lightning presented in the previous chapter (*SpecificationI*) is a real-time specification and we will define a corresponding time-optimized specification (*SpecificationI*^{time}).

The approach of constructing the time-optimized specification is general and can directly be transferred to other use-cases of real-time specifications in TLA$^+$. Thereby, the construction of the time-optimized specification and the proof can be reused beyond this thesis.

4.2. Decomposition of a Payment Channel Network Into Single Payment Channels

Besides the modeling of time, model checking a specification of a network of payment channels is complex because the size of the state space of such specifications is exponential in the number of channels. To facilitate model checking that a lower-level specification

implements a higher-level specification, we utilize the fact that all payment channels are specified in the same way. We decompose a specification of a network of payment channels into single payment channels that are described by a single channel specification. The single payment channel specifications are defined so that they specify all possible behaviors of a payment channel that is combined in a network with other payment channels. We define a refinement mapping from the lower-level single channel specification to the higher-level single channel specification and model check that the lower-level single channel specification implements the higher-level single channel specification. By showing that the payment channels described by the higher-level single channel specification can be combined to a higher-level specification of a network of payment channels, we prove that the lower-level specification of a network of payment channels implements the higher-level specification.

Example 4.10. Figure 4.22 on page 125 visualizes a specific instance of this approach for the refinement between $SpecificationI^{time}$ and *SpecificationII* for which the single channel specifications $SpecificationI^{time,\ single}$ and $SpecificationII^{single}$ are defined. In the case shown in Fig. 4.22, $SpecificationI^{time}$ is the lower-level specification and *SpecificationII* is the higher-level specification. $SpecificationI^{time}$ and *SpecificationII* both define the behavior of a network of payment channels. We define $SpecificationI^{time,\ single}$ that defines the behavior of any single payment channel that is part of the network specified in $SpecificationI^{time}$. Analogously, we define the single channel specification $SpecificationII^{single}$ for *SpecificationII*. As we describe later in Section 4.3.3, we define a refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$ and model check that the refinement mapping is correct. We prove that $SpecificationI^{time}$ implements *SpecificationII* by showing that every channel that has been mapped from $SpecificationI^{time}$ to $SpecificationI^{time,\ single}$ and from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$ can be combined with other channels and the resulting behavior is a behavior of *SpecificationII*.

A first attempt to define a single channel specification could be to restrict the size of the payment channel network in a specification of a network of payment channels to two users and a single payment channel. Such a specification, however, would only describe behaviors of a channel in a single-hop payment but would not describe behaviors of a channel that forwards a multi-hop payment. To ensure that a single channel specification describes all behaviors of a channel that is combined with other payment channels, a single channel specification must model the effects that other channels have on the modeled payment channel. These effects are, for example, that a user retrieves a payment that should be forwarded or that a user retrieves a preimage for a payment that has been forwarded. To model such effects, we add a dedicated module to each single payment channel specification that models the channel's environment. With the environment module, a single channel specification describes all possible behaviors of a payment channel in a specification of a network of payment channels.

We apply this decomposition approach to show that *SpecificationII* abstracts *SpecificationI* and to show that *SpecificationIII* abstracts *SpecificationII*. In contrast to the time abstraction (see Section 4.1), we do not write a general proof but use the same proof approach to write individual proofs for both abstractions. To apply the approach to show that a higher-level

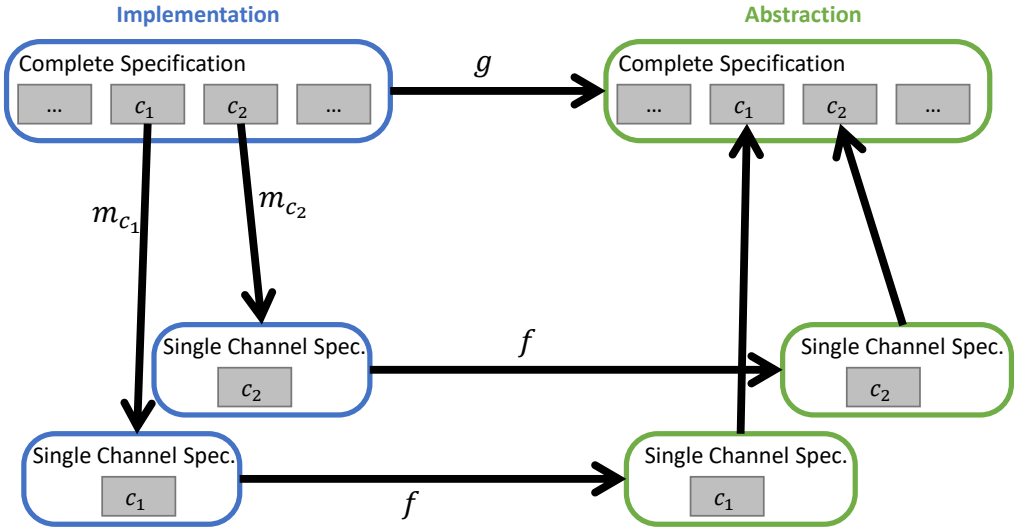

Figure 4.19.: Overview of the decomposition proof step approach. Our goal is to show that a specification (top left) is an implementation of a more abstract specification (top right). We define the refinement mapping g from the specification of the implementation to the specification of the abstraction. The mapping g uses a mapping f that maps each channel of the implementation specification to a channel of the abstract specification. To show that the mapping g is a refinement mapping, we define single channel specifications (bottom) and a mapping m_c that maps each channel c of the complete specification of the implementation to a single channel specification. In a next step, we model check that the mapping f is a refinement mapping from the single channel specification of the implementation to the abstract single channel specification. Finally, we prove that the abstract single channel specifications can be combined into a network of payment channels.

specification (*SpecificationII* or *SpecificationIII* resp.) abstracts a lower-level specification (*SpecificationI* or *SpecificationII* resp.), we take the following steps as depicted in Fig. 4.19:

1. We define single channel specifications for the lower-level specification and the higher-level specification. These single channel specifications include an environment module.

2. We define a mapping m_c from the lower-level specification to the corresponding single channel specification that maps a specific channel c to the single channel specification. We present and prove a lemma that states that it holds for each payment channel in the lower-level specification that the mapping m_c maps a behavior of the lower-level specification of a network of payment channels to a behavior of the corresponding single channel specification. This implies that the environment module abstracts all effects that other channels can have on the modeled channel.

3. We define a mapping f from the lower-level single channel specification to the higher-level single channel specification. We present a lemma that states that the mapping f is a refinement mapping. We verify the correctness of this lemma using model checking.

4. We define how single channels in the higher-level specification are combined with other channels to form a higher-level specification of a network of channels. We

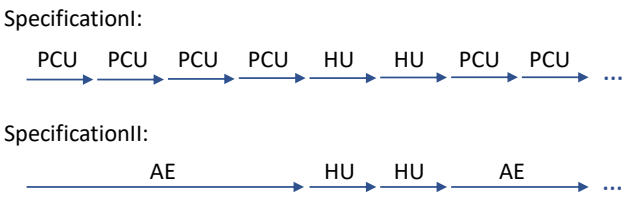

Figure 4.20.: Sketch of an exemplary behavior of *SpecificationI* and the corresponding behavior of *SpecificationII*. The abbreviations PCU, HU, and AE stand for *PaymentChannelUser*, *HTLCUser*, and *AbstractEnforcement* respectively. The sketch shows that multiple steps of the module *PaymentChannelUser* are mapped to a single step of the module *AbstractEnforcement* while steps of the module *HTLCUser* in *SpecificationI* are mapped to the same steps in *SpecificationII*.

present and prove a lemma that states that every step of a channel c in the higher-level single channel specification for channel c is also possible when the channel c is combined with other payment channels in the higher-level specification.

5. Finally, we present a theorem that states that the higher-level specification abstracts the lower-level specification and we prove the theorem based on the previously presented lemmas.

Having discussed the time-optimization and the decomposition approach, we will discuss in the following sections the concrete abstractions of the stepwise refinement. We apply the time-optimization and the decomposition approach to check these abstractions.

4.3. Abstraction of Enforcement Layer

The first abstraction of the stepwise refinement is to abstract the payment channel implementation on the enforcement layer, i.e., the module *PaymentChannelUser* that handles all actions necessary to enforce the state of an HTLC on the blockchain. In this section, we describe the idea behind the abstraction of *SpecificationII* and how we use the time abstraction and the decomposition approach to show that *SpecificationI* implements *SpecificationII*.

The specification of Lightning in *SpecificationI* is composed mainly of the modules *PaymentChannelUser* and *HTLCUser*. The actions for making payments are primarily specified in the module *HTLCUser*. The actions of the module *HTLCUser* require updates of the enforcement layer, e.g., an HTLC that has been added with state NEW must advance to the state COMMITTED so that the HTLC can be fulfilled. These updates of the enforcement layer are specified by actions of the module *PaymentChannelUser*. Many steps of the module *PaymentChannelUser* are internal updates of the payment channel and affect only the payment channel itself but do not affect whether the actions of the module *HTLCUser* or the actions of the module *PaymentChannelUser* for other payment channels are enabled. For example, during the opening of a payment channel, several steps need to be taken by actions of the module *PaymentChannelUser* until the channel is open and an action of the module *HTLCUser* becomes enabled. Also, during an update of a channel, it takes several

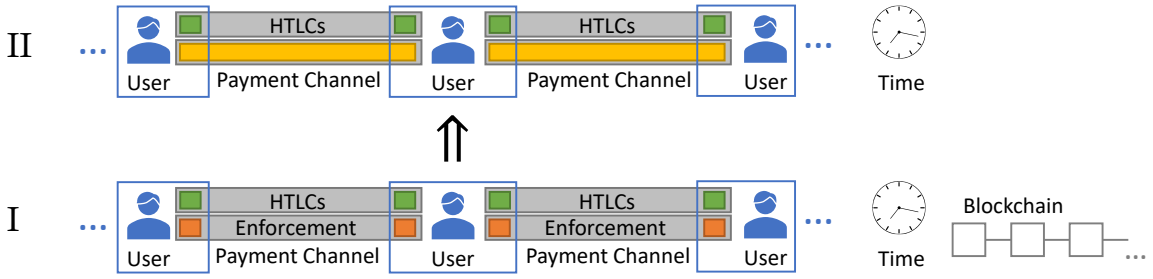

Figure 4.21.: System overview of *SpecificationI* and *SpecificationII*. In *SpecificationI*, there is for each payment channel an instance of the module *HTLCUser* (green boxes) on the HTLC layer for each user and an instance of the module *PaymentChannelUser* (orange boxes) on the enforcement layer for each user. Each step of *SpecificationI* is a step of one of the instances of the modules *HTLCUser* or *PaymentChannelUser* or the blockchain (module *LedgerTime*). In *SpecificationII*, the two instances of the module *PaymentChannelUser* in each channel are replaced by a single instance of the module *AbstractEnforcement* (yellow box). The module *AbstractEnforcement* provides a more abstract implementation of the enforcement layer that does not use a blockchain.

steps of the module *PaymentChannelUser* until the state of an HTLC has changed from NEW to COMMITTED so that the HTLC could be fulfilled or timed out by an action of the module *HTLCUser*. To reduce the specification's state space, these intermediate states of the module *PaymentChannelUser* with only local effects can be abstracted. In this section, we present *SpecificationII* in which such sets of consecutive steps of *PaymentChannelUser* are replaced by a single step of the module *AbstractEnforcement* (see Fig. 4.20) and we verify that *SpecificationI* implements *SpecificationII*.

We put this idea in the context of the system overview (see Fig. 4.21). The idea of the abstraction presented in this section is to replace the two instances of the module *PaymentChannelUser* in a payment channel (orange boxes in Fig. 4.21) by an instance of the module *AbstractEnforcement* (yellow box in Fig. 4.21). The module *AbstractEnforcement* must change the variables that are accessed by the modules *PaymentChannelUser* and *HTLCUser* in exactly the same way as the module *PaymentChannelUser* changes those variables. From the perspective of an instance of the module *HTLCUser*, it must be indistinguishable whether the enforcement layer is implemented by two instances of the module *PaymentChannelUser* or a single instance of the module *AbstractEnforcement*. Thus, for each step of the module *PaymentChannelUser* that interacts with the module *HTLCUser* by updating a shared variable to a state that is relevant for an action of the module *HTLCUser*, there must be a corresponding step in the module *AbstractEnforcement*. Internal steps of the module *PaymentChannelUser* that do not change any variables that are relevant for other modules can be abstracted by the module *AbstractEnforcement* by stuttering steps.

The approach to show that *SpecificationI* implements *SpecificationII* is depicted in Fig. 4.22. We first create $SpecificationI^{time}$ as a time-optimized version of *SpecificationI*. We apply Theorem 1 (see page 112) to show that *SpecificationI* implements $SpecificationI^{time}$. To this end, we prove in Section 4.3.1 that *SpecificationI* and $SpecificationI^{time}$ fulfill the assumptions of Theorem 1. In Section 4.3.2, we introduce the module *AbstractEnforcement* by giving an overview of the module, the ideas behind its definition, and by briefly presenting

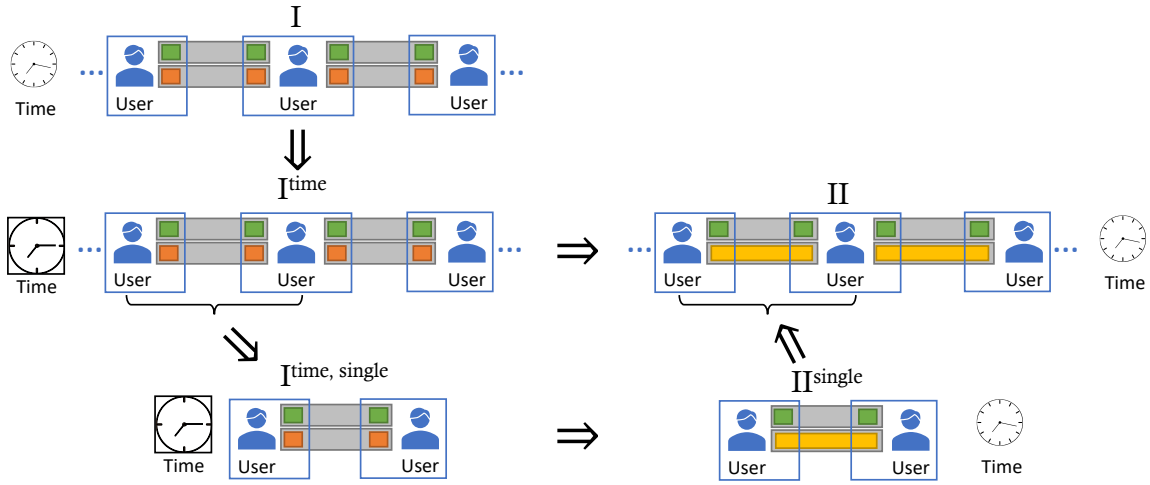

Figure 4.22.: Approach to check the abstraction from *SpecificationI* to *SpecificationII*. We use the time abstraction (see Section 4.1) to show that *SpecificationI* implements $SpecificationI^{time}$ and, then we decompose $SpecificationI^{time}$ and show that each single channel specification $SpecificationI^{time,\ single}$ implements the single channel specification $SpecificationII^{single}$. The composition of payment channels specified by $SpecificationII^{single}$ is a payment channel network as specified by *SpecificationII*.

each action of the module *AbstractEnforcement*. In Section 4.3.3, we give an overview of the proof that $SpecificationI^{time}$ implements *SpecificationII*. This proof uses the decomposition approach to model check that $SpecificationI^{time}$ implements *SpecificationII*.

4.3.1. Application of Theorem 1 for *SpecificationI*

In this subsection, we apply Theorem 1 to show that *SpecificationI* implements $SpecificationI^{time}$. To this end, we take the following steps:

1. We show that *SpecificationI* is a real-time specification.
2. We define a corresponding time-optimized specification $SpecificationI^{time}$.
3. We prove that the assumptions Assumptions 4.1 to 4.3 hold for *SpecificationI* and $SpecificationI^{time}$ so that Theorem 1 can be applied.

4.3.1.1. *SpecificationI* Is a Real-Time Specification

To specify a real-time system according to the definition in Section 4.1.1, one has to specify

- a set of clocks $\mathcal{X}$,
- an initial value I^x for each clock $x \in \mathcal{X}$,
- a mapping B^x for each clock $x \in \mathcal{X}$ that assigns a set of time bounds to each state, and

- an $InnerNext$ action that defines all steps of the system that do not advance time.

SpecificationI implements such a real-time specification with the set of clocks $\mathcal{X}$ that contains the clock *LegerTime* and a clock *TxAge[id]* for the id of each transaction in the set *LedgerTx*. Formally: $\mathcal{X} = \{LedgerTime\} \cup \{TxAge[id] : id \in LedgerTx\}$. The initial values of the clocks are: For the clock *LedgerTime*, the initial value is $I^{LedgerTime} = 1$ and for the clocks in *TxAge*, the initial value is 0. The mapping of time bounds B^x is defined in *SpecificationI* by the operator *TimeBounds* that is defined as the union of the operators with the name *TimeBounds* of the modules *PaymentChannelUser* and *HTLCUser*. The *InnerNext* action of *SpecificationI* allows steps of the modules *PaymentChannelUser* or *HTLCUser* or a step of *WithdrawBalancesAfterChannelClosed* (see Fig. 3.9 on page 54). Thereby, it follows that *SpecificationI* is a real-time specification.

4.3.1.2. Definition of *SpecificationI*^{time}

In Section 4.1.3, we defined a corresponding time-optimized system for such a specification of a real-time system. To apply the proof in Section 4.1.5, one has to define the set of relaxed zone time points $relZTP^x$ for each clock $x \in \mathcal{X}$. The proof relies on an assumption on the definition of the time bounds B^x (Assumption 4.1) and on two assumptions on the definition of the relaxed zone time points $relZTP^x$ (Assumptions 4.2 and 4.3). In the following, we prove that definition of time bounds in *SpecificationI* and the definition of the relaxed zone time points in *SpecificationI*^{time} fulfill these assumptions.

4.3.1.3. Assumptions 4.1 to 4.3 Hold for *SpecificationI* and *SpecificationI*^{time}

We start by proving that the time bounds specified in *SpecificationI* meet Assumption 4.1.

Lemma 4. *In SpecificationI it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T_d^y(s))$$

We prove Lemma 4 in the appendix in Section A.3.5. The idea of the proof is to show that the variables that model the clocks do not appear in the definitions of the time bounds. Therefore, it follows that the time bounds do not depend on the values of the clock variables.

We prove that $relZTP^x$ meets Assumption 4.2 as stated by Lemma 5. The set of relaxed zone time points $relZTP^{LedgerTime}$ for the clock *LedgerTime* is defined by the operator *relZTP* in *SpecificationI*^{time} and the set of relaxed zone time points $relZTP^{TxAge}$ for the clocks in *TxAge* is defined by the set of relative timelocks used in each transaction's outputs. □

Lemma 5. *The definition of relaxed zone time points in $relZTP^x$ of SpecificationI*^{time} *meets Assumption 4.2, i.e. it holds that* $\forall x \in \mathcal{X}, s \in \hat{\Sigma} : ZTP^x(s) \subseteq relZTP^x(s)$.

$$
\begin{array}{l}
SendSignedCommitment \triangleq \\
\quad \wedge \text{ LET } HTLCsToAdd \triangleq \{htlc \in Vars.OutgoingHTLCs) : \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ htlc.state = \text{“NEW”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ LedgerTime < htlc.absTimelock\} \\
\qquad\quad HTLCsToRemove \triangleq \{htlc \in Vars.IncomingHTLCs) : \\
\qquad\qquad\qquad\qquad\qquad\qquad htlc.state = \text{“FULFILLED”}\} \\
\qquad \text{IN } \ Vars' = [\, Vars \text{ EXCEPT} \\
\qquad\qquad\qquad !.OutgoingHTLCs = (@ \setminus HTLCsToAdd) \cup \\
\qquad\qquad\qquad\qquad \{[htlc \text{ EXCEPT } !.state = \text{“SENT-COMMIT”}] : \\
\qquad\qquad\qquad\qquad\qquad\qquad htlc \in HTLCsToAdd\}, \\
\qquad\qquad\qquad !.IncomingHTLCs = (@ \setminus HTLCsToRemove) \cup \\
\qquad\qquad\qquad\qquad \{[htlc \text{ EXCEPT } !.state = \text{“SENT-REMOVE”}] : \\
\qquad\qquad\qquad\qquad\qquad\qquad htlc \in HTLCsToRemove\} \\
\qquad\qquad\quad] \\
\quad \wedge \ \ldots
\end{array}
$$

Figure 4.23.: Partial definition of the action for sending a signed commitment serving as an example of the proof of Lemma 5. The shown code excerpt defines a set $HTLCsToAdd$ as the set of all outgoing HTLCs that are in state NEW and whose absolute timelock is greater than the value of *LedgerTime* and a set $HTLCsToRemove$ as the incoming HTLCs that are in state FULFILLED. The state of the outgoing HTLCs to be added is updated to SENT-COMMIT by defining the new value of *Vars.OutgoingHTLCs* as the old value (@) minus the HTLCs in $HTLCsToAdd$ plus a set of all HTLCs in $HTLCsToAdd$ whose state is updated to SENT-COMMIT. The state of the fulfilled HTLCs is updated analogously to SENT-REMOVE.

Informally, Lemma 5 means that the definition of the relaxed zone time points $relZTP^x(s)$ that is explicitly defined in *SpecificationI* includes all zone time points. We prove the lemma in Section A.3.6 and sketch the proof idea here.

The definition of zone time points in Definition 11 defines a zone time point as the clock value of a reachable state at which a $Next_S$ step of the system S is possible but the same step is not possible at the directly preceding point in time. $Next_S$ in *SpecificationI* is defined as the action $NextI$ and we use that name in the following. The proof is structured into several cases by looking at each subaction of $NextI$ separately. For each subaction A of $NextI$, we show that, if there exists a reachable state at which an A step is possible at point in time d but the same step is not possible at the directly preceding point in time $d-1$, then the point in time d is contained in the set $relZTP^x(s)$. To find such points in time, we analyze for each action A and clock x how the action A depends on the clock x.

Here we give a simplified example. Consider the partial definition of a simplified version of the action *SendSignedCommitment* given in Fig. 4.23. The action updates the state of all outgoing HTLCs that are in state NEW and whose absolute timelock is greater than the clock *LedgerTime* as well as the state of incoming HTLCs that are in state FULFILLED. Depending on the value of *LedgerTime*, the steps described by the action *SendSignedCommitment* update the state of a different set of HTLCs. Consider as an example a state in which three outgoing HTLCs exist in state NEW with the absolute timelocks 4, 8 and 9 and there exists also an incoming HTLC in state FULFILLED. Figure 4.24 shows that depending on the value of *LedgerTime* a step of the action *SendSignedCommitment* adds a different set of outgoing

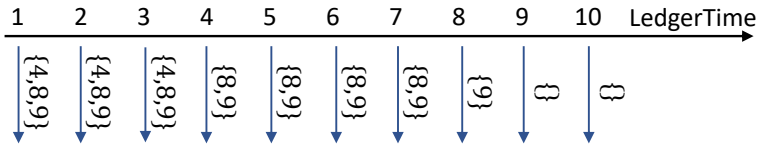

Figure 4.24.: Sketch showing which outgoing HTLCs might be added depending on the current value of *LedgerTime*. This is based on an exemplary scenario in which there exist two outgoing HTLCs exist in state NEW with the absolute timelocks 4, 8 and 9 and an incoming HTLC in state FULFILLED. The vertical arrows represent steps of the simplified action *SendSignedCommitment* (see Fig. 4.23) at different points in time. Each arrow is labeled by the absolute timelocks of outgoing HTLCs that are added in the step. In each step the state of the incoming HTLC that is in state FULFILLED is updated (not shown in the sketch). Therefore, steps of the action *SendSignedCommitment* are also possible if no outgoing HTLC may be added.

HTLCs. If the value of *LedgerTime* is below 4, then the state of all three outgoing HTLCs is updated. If the value of *LedgerTime* is between 4 and 7, only the state of the two outgoing HTLCs with the absolute timelocks 8 and 9 is updated. If the value of *LedgerTime* equals 8, only the state of the outgoing HTLC with the absolute timelock 9 is updated. If the value of *LedgerTime* is greater than or equal to 9, no outgoing HTLC is added but a step of the action *SendSignedCommitment* is still possible because of the incoming HTLC that is removed. In Fig. 4.24, it can be seen that at time points 4, 8 and 9 (the absolute timelocks of the HTLCs) a step is possible that is not possible at the directly preceding point in time. Thus, these time points are zone time points for all states s from which a state is reachable in which these three outgoing HTLCs can be added (as shown in Fig. 4.24. These states s are all states that are reachable by the initial states except states in which the three outgoing HTLCs have already been added or have been aborted and, thus, are not in state NEW and cannot be added. To generalize from the example, the definition of relaxed zone time points $relZTP^{LedgerTime}(s)$ must contain the absolute timelock of all HTLCs that are in state NEW or can be in state NEW in a reachable state. The definition of relaxed zone time points $relZTP^{LedgerTime}(s)$ in $SpecificationI^{time}$ fulfills this requirement by including the absolute timelock of all outgoing HTLCs in state NEW and, for each payment for which an HTLC has not yet been created, the timelocks of outgoing HTLCs that will be created along the payment's route[5].

In the proof, we consider every subaction A of $NextI$ and clock x and, if the subaction depends on clock x, we determine the zone time points induced by the action A and show that these zone time points are included in the definition of relaxed zone time points $relZTP^x$ as defined in $SpecificationI^{time}$. Thereby, we prove that the relaxed zone time points $relZTP^x$ contain all zone time points which proves Lemma 5. □

Next, we prove that *SpecificationI* fulfills Assumption 4.3, i.e. that for a $NextI$ step from state s to state t it holds for all clocks $x \in \mathcal{X}$ that $relZTP^x(t) \subseteq relZTP^x(s)$.

[5] This is possible because the timelocks along a payment's route can be anticipated because they are chosen deterministically.

Lemma 6. *For any two states s, t of SpecificationI and all clocks $x \in \mathcal{X}$ (i.e., LedgerTime and TxAge), it holds that*

$$\mathit{NextI} \Rightarrow relZTP^{x\prime} \subseteq relZTP^{x}$$

We prove Lemma 6 in the appendix in Section A.3.7. The set $relZTP^x$ is defined in *SpecificationI* as the union of smaller sets. In the proof, we show for each of these smaller sets Z that it holds that $Z' \subseteq Z$ during a step of $NextI$. We also differentiate the subactions of $NextI$ into several cases to facilitate the proof.

To give an example, one of the subsets that constitute the set $relZTP^x$ is the set of absolute timelocks of all HTLCs that are in states $H_1 = \{$ NEW, RECV-COMMIT, PENDING-COMMIT, SENT-COMMIT, COMMITTED $\}$. For steps of all actions that do not create new HTLCs, it holds that $H_1' \subseteq H_1$ because the actions either do not change the state of HTLCs or advance the state of HTLCs as specified in Section 3.3.2.7. The definition of how the state of an HTLC can be advanced implies that the new state of an HTLC can only be a state that is in the set H_1 if the old state of the HTLC was already in the set H_1. □

Finally, we show that *SpecificationI* implements $\mathit{SpecificationI}^{time}$.

Theorem 2.

$$\mathit{SpecificationI} \Rightarrow \mathit{SpecificationI}^{time}$$

Proof: From Lemmas 4 to 6 it follows that *SpecificationI* fulfills the assumptions Assumptions 4.1 to 4.3. Therefore, we can apply Theorem 1 to *SpecificationI*. Because $\mathit{SpecificationI}^{time}$ is defined according to the definition of the time-optimized specification $Spec_{\hat{S}}$ in Section 4.1.3, this implies that *SpecificationI* implements $\mathit{SpecificationI}^{time}$. □

4.3.2. *SpecificationII* and Module *AbstractEnforcement*

In comparison to *SpecificationI*, *SpecificationII* uses the module *AbstractEnforcement* instead of the module *PaymentChannelUser* to model the behavior of payment channels. While the module *PaymentChannelUser* models the actions of a single user, the module *AbstractEnforcement* abstracts the actions of the two users in a payment channel. In this subsection, we explain the ideas behind *SpecificationII* and present the module *AbstractEnforcement* in more detail.

The module *AbstractEnforcement* abstracts from the specific protocol steps and replaces them by more abstract steps. Protocol details such as messages that are exchanged between the parties and blockchain transactions are not included in the specification of the module *AbstractEnforcement*. Instead, the module *AbstractEnforcement* specifies how the variables of a payment channel that are relevant for multi-hop payments are changed. In particular, the states of HTLCs are updated according to Fig. 4.25 and only states are included in the specification that are also relevant for actions of the module *HTLCUser*. The intermediate HTLC states that are used in *SpecificationI* (see Fig. 3.13 on page 66) are skipped by the

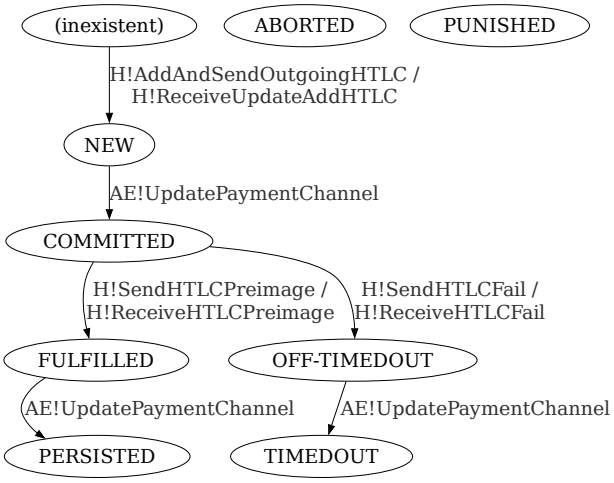

Figure 4.25.: Flow chart of HTLC states in *SpecificationII*. The actions with the prefix H! are actions of the *HTLCUser* module (see Section 3.3.4); those prefixed with AE! are actions of the *AbstractEnforcement* module. The flow chart shows how the states of an HTLC are updated off-chain. The states ABORTED and PUNISHED are only reached when the channel is closed.

module *AbstractEnforcement* because these intermediate states are not relevant for actions of the module *HTLCUser*. Most of the actions of the module *AbstractEnforcement* update the state of both users in the payment channel simultaneously, e.g., the state of an HTLC changes from NEW to COMMITTED in the state of both users in the same step.

The decision which actions to model in the module *AbstractEnforcement* is based on the following reasoning. The actions of the module *AbstractEnforcement* should abstract as many consecutive steps of the module *PaymentChannelUser* as possible. However, each step of the module *PaymentChannelUser* that describes a change that affects whether a step of an action of another module is possible must be mapped to a step of the module *AbstractEnforcement* that has the same effect. During opening a payment channel and during updating a channel, many steps can be abstracted by a single step. However, a step in which an HTLC is fulfilled on the blockchain changes the preimages known by a user and, because these preimages are relevant for actions of the module *HTLCUser*, the step must be described by an action of the module *AbstractEnforcement*. Following these design ideas, the module *AbstractEnforcement* has the following actions.

The action *OpenPaymentChannel* describes that the protocol state of both users is changed in a single step from `init` to `idle` and the balances are updated so that the funder's external balance is reduced and the channel balances are set so that the funder owns the entire capacity of the channel.

The action *UpdatePaymentChannel* abstracts the users' exchange of signatures for new commitment transactions and the revocation of the old commitment transactions. In a step described by the action *UpdatePaymentChannel*, the state of HTLCs can be advanced from NEW to COMMITTED, from FULFILLED to PERSISTED, and from OFF-TIMEDOUT to TIMEDOUT.

The action *UpdatePaymentChannel* contains several checks to allow only those updates of HTLC states that are possible considering the users' balances.

The closing of a channel is described by the action *SetOnChainHTLCsAndCheater* which stores which HTLCs are included in the commitment transaction that has closed the channel and whether the channel was closed honestly, i.e., with the latest commitment transaction. The set of HTLCs that are included in the commitment transaction is relevant because these HTLCs can be fulfilled on the blockchain which is described by the following actions.

The actions *FulfillIncomingHTLCsOnChain* and *NoteFulfilledHTLCsOnChain* describe that a user fulfills an incoming HTLC on the blockchain and that the other user reads the preimage from the blockchain. The users' balances are adjusted and the payment's state is updated if the fulfilling user is the receiver of the corresponding payment or the user receiving the preimage is the sender of the corresponding payment. In contrast to the action *OpenPaymentChannel* and *UpdatePaymentChannel*, these steps abstract only a single step of the module *PaymentChannelUser*. The reason for this is that the steps in which a preimage is published and in which a preimage is learned, affect other payment channels and, thus, the effect of these steps must be directly visible for other modules.

The action *CommitHTLCsOnChain* describes that both users have noted the commitment transactions on the blockchain and have updated their states accordingly, i.e. the state of an HTLC can be advanced to ABORTED if the HTLC was in state NEW and is not included in the commitment transaction and to COMMITTED if the HTLC is included in the commitment transaction. The definition also includes the option that HTLCs are fulfilled in the same step as defined in the action *FulfillHTLCsOnChain* whose description follows.

The action *FulfillHTLCsOnChain* models that an HLC is fulfilled on the blockchain and the state of the HTLC advances in both user's states to PERSISTED. As above for the actions *FulfillIncomingHTLCsOnChain* and *NoteFulfilledHTLCsOnChain*, the users' balances and payment states are adjusted accordingly. The difference of the action *FulfillHTLCsOnChain* to these actions is that the action *FulfillHTLCsOnChain* describes steps in which the variables of both users are changed in the same step.

The final closing of the abstracted channel is modeled by the action *ClosePaymentChannel* which models that the states of all HTLCs are set to a final HTLC state, i.e. ABORTED, PERSISTED, TIMEDOUT, or PUNISHED. The HTLCs are advanced to state PUNISHED if and only if one of the users has cheated. The balances of the users are adjusted if HTLCs are fulfilled in this step or if one of the users has cheated.

As the module *PaymentChannelUser*, the module *AbstractEnforcement* also needs to specify time bounds for actions that are urgent. To simplify the abstraction of steps of *PaymentChannelUser*, the module *AbstractEnforcement* uses a slightly different approach than the module *PaymentChannelUser*. In principle, there are two types of steps for which urgency is required in *SpecificationI* where the module *PaymentChannelUser* is used. The first type are, for example, steps in which a user fulfills an incoming HTLC within the required time frame if the user is honest and knows the preimage for fulfilling the HTLC. Otherwise the user risks that the HTLC is timed out and does not receive the incoming

payment. The other type are steps for persisting the state of the payment channel on the blockchain. For example, when a commitment transaction is published that contains an HTLC that an honest user has already fulfilled, the honest user should fulfill the HTLC also on the blockchain by publishing the HTLC success transaction. Otherwise the honest user would risk that the other party spends the HTLC output in the commitment transaction by a timeout transaction after the HTLC's timelock has passed. While the first type of urgency (fulfilling an incoming HTLC if the preimage is known) is also required in the module *AbstractEnforcement*, the second type of urgency does not need to be modeled as urgency in the module *AbstractEnforcement*. Instead, the actions of the module *AbstractEnforcement* specify that an HTLC that has been fulfilled cannot be timed out later if the user who has fulfilled the HTLC is honest.

While there are several variables used by the module *PaymentChannelUser* that are not used by the module *AbstractEnforcement* because the module *AbstractEnforcement* is more abstract, the module *AbstractEnforcement* uses the additional variable *LatePreimages* that is not used by the module *PaymentChannelUser*. The variable *LatePreimages* stores preimages that are received for an outgoing HTLC after the HTLC's timelock plus the forward delta[6]. An honest user times out an outgoing HTLC before the HTLC's absolute timelock plus the forward delta to ensure that the outgoing HTLC is timed out before the corresponding incoming HTLC can be timed out. However, it might be that the user still receives the preimage for the timed out HTLC after the HTLC's absolute timelock plus the forward delta, for example, because the message containing the preimage was delayed for more than the forward delta. Using the variable *LatePreimages*, it can be distinguished in the module *AbstractEnforcement* whether a preimage was received in time or whether a preimage was received too late. If an honest user who forwards a payment has received the corresponding preimage in time, the module *AbstractEnforcement* specifies that the incoming HTLC will eventually be fulfilled. Otherwise, if the preimage is contained in the variable *LatePreimages*, the incoming HTLC may be timed out.

4.3.3. *SpecificationI*^{time} Implements *SpecificationII*

Having specified *SpecificationII*, our goal is to prove that *SpecificationI*^{time} implements *SpecificationII*. The main idea of the proof is to show that every step in *SpecificationI*^{time} can be mapped to a step in *SpecificationII* because the module *AbstractEnforcement* abstracts the actions of the module *PaymentChannelUser*. To validate this abstraction, we define a refinement mapping g from *SpecificationI*^{time} to *SpecificationII* and prove the correctness of this refinement mapping.

A challenge for the proof is that the refinement mapping g is too complicated to be proven directly and that *SpecificationI*^{time} models an entire network of interacting payment channels and, thus, the specification's state space is too large for model checking. To facilitate

[6] The forward delta is the difference between the timelocks of the incoming HTLC and the outgoing HTLC of a forwarded payment (see Section 2.3.5).

model checking, we use the decomposition approach as presented in Section 4.2. Concretely, we utilize the fact that all payment channels modeled in *SpecificationI*^{time} are specified in the same way using the modules *PaymentChannelUser* and *HTLCUser* and the same holds for the payment channels in *SpecificationII* specified using the modules *AbstractEnforcement* and *HTLCUser*. Thus, the specifications *SpecificationI*^{time} and *SpecificationII* can be seen as compositions of multiple single channel specifications. Because all of the single channel specifications in one of these specifications are equal, it suffices to check that a single payment channel in *SpecificationI*^{time} implements a single payment channel in *SpecificationII*.

We realize such an approach by specifying the specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} that model only a single payment channel of *SpecificationI*^{time} and *SpecificationII* respectively. We define a refinement mapping f that maps a state of *SpecificationI*$^{time,\ single}$ to a state of *SpecificationII*^{single}. The state space of *SpecificationI*$^{time,\ single}$ is small enough so that the correctness of the refinement mapping f can be checked for small models using model checking (see Section 4.3.3.2).

To ensure that the result of checking the payment channel abstraction between the single channel specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} can be transferred to the multi-channel specifications *SpecificationI*^{time} and *SpecificationII*, it must hold that every behavior of a channel in *SpecificationI*^{time} is also possible in *SpecificationI*$^{time,\ single}$. Because a payment channel c in *SpecificationI*^{time} can be influenced by other payment channels and users, the variables of the payment channel c in *SpecificationI*^{time} might change in ways that are not modeled by the actions of the modules *HTLCUser* and *PaymentChannelUser* for steps of the channel c. To model these effects that the environment of the channel c has on channel c, we specify the module *ChannelEnvironment* that abstracts all effects that a channel's environment can have on a channel. Such effects are, for example, that an HTLC is forwarded or that a preimage for an incoming HTLC is received. Specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} include the module *ChannelEnvironment* to ensure that all behaviors that are possible for a channel in *SpecificationI*^{time} are also possible for the channel in *SpecificationI*$^{time,\ single}$. We present the module *ChannelEnvironment* in more detail in Section 4.3.3.1.

The structure of the proof that the mapping g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII* is shown in Fig. 4.26. The basic idea for defining the refinement mapping g is to map each channel in *SpecificationI*^{time} to a channel in *SpecificationI*$^{time,\ single}$, then use the refinement mapping f to map the channel to an abstracted channel specified by *SpecificationII*^{single}, and, finally, combine the mapped channels again to a network of channels in *SpecificationII*. To this end, we first define a mapping m_c that maps a channel c from *SpecificationI*^{time} to a state of *SpecificationI*$^{time,\ single}$. We prove that the module *ChannelEnvironment* correctly abstracts all steps that affect a channel in *SpecificationI*^{time} and are not a step of the channel itself (see Lemma 7 on page 141). Then, we prove that the mapping m_c correctly maps each state of a channel in *SpecificationI*^{time} to a state of *SpecificationI*$^{time,\ single}$ (see Lemma 8). We use model checking to check that the mapping f is a refinement mapping from *SpecificationI*$^{time,\ single}$ to *SpecificationII*^{single} (see Lemma 9). Further, we prove that a step of a channel in *SpecificationII*^{single} is also possible if the

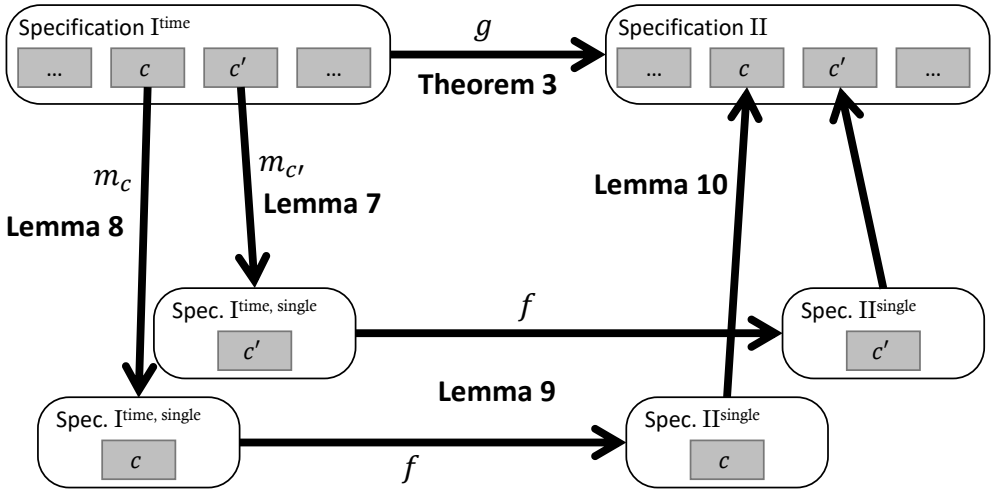

Figure 4.26.: Overview of the proof that *SpecificationI*^{time} implements *SpecificationII*. To prove Theorem 3, we show that the mapping g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII*. For this proof, we prove the following: Each step in *SpecificationI*^{time} in channel c is mapped to a valid step of an instance of *SpecificationI*$^{time,\ single}$ for channel $c' \neq c$ (Lemma 7) by the mapping $m_{c'}$ and, by the mapping m_c to a valid step of an instance of *SpecificationI*$^{time,\ single}$ for channel c (Lemma 8). We use model checking to check that the mapping f is a refinement mapping from *SpecificationI*$^{time,\ single}$ to *SpecificationII*^{single} (Lemma 9). We prove that a step of channel c in *SpecificationII*^{single} is also possible if the channel is combined with other channels in *SpecificationII* (Lemma 10).

channel is combined with other channels in *SpecificationII* (see Lemma 10). Finally, we prove Theorem 3 to show that the mapping g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII* based on Lemmas 7 to 10. See Fig. 4.27 for an overview. The mapping g works as follows: In a first step, the mapping g defines for each channel c in *SpecificationI*^{time} a state of *SpecificationI*$^{time,\ single}$. This state of *SpecificationI*$^{time,\ single}$ is mapped to a state of *SpecificationII*^{single} using the refinement mapping f. The mapping g assigns to a state of *SpecificationI*^{time} a state of *SpecificationII* with the channel variables of *SpecificationII*^{single} for each channel c and the user and general variables of the given state of *SpecificationI*^{time}.

This section continues with a presentation of the single channel specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} in Section 4.3.3.1 followed by a presentation of the refinement mapping f between these specifications in Section 4.3.3.2. In Section 4.3.3.3, we prove that channels specified by *SpecificationII*^{single} can be combined to a payment channel network as specified by *SpecificationII*. Subsequently, Theorem 3 that *SpecificationI*^{time} implements *SpecificationII* is proven in Section 4.3.3.4.

4.3.3.1. Channels in *SpecificationI*^{time} Implement *SpecificationI*$^{time,\ single}$

To facilitate checking that *SpecificationI*^{time} refines *SpecificationII*, we introduce two variations of these specifications that are for a single channel only. These specifications are *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single}. In general, both single-channel specifications have the same variables as their full counterparts but, while *SpecificationI*^{time} and

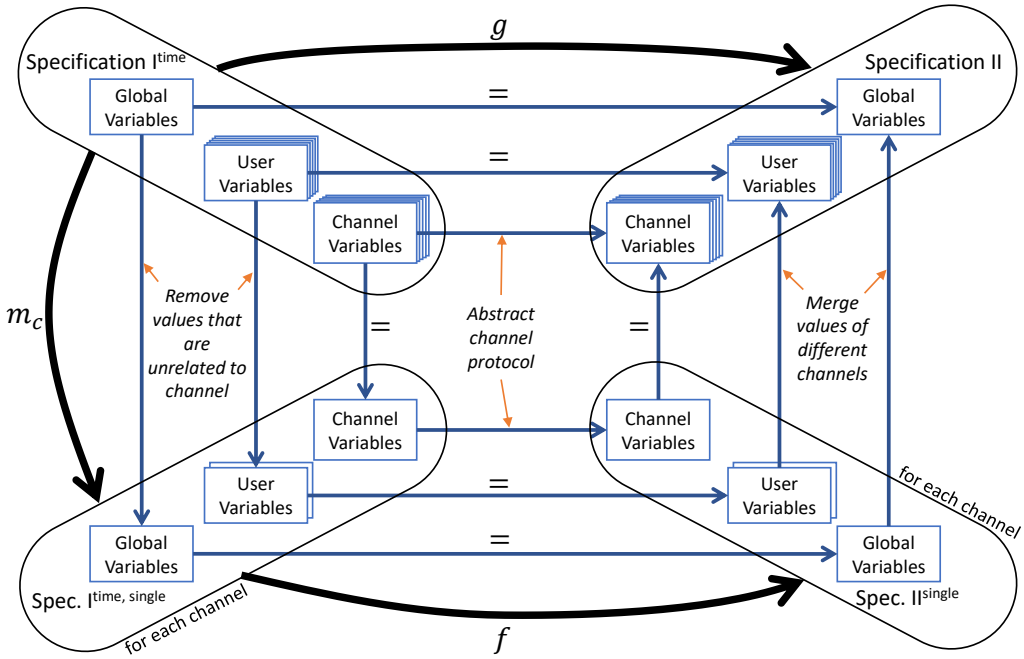

Figure 4.27.: Overview of the mappings between the specifications used in the decomposition approach. Each specification's states consist of a set of global variables, user variables, and channel variables. While specifications *SpecificationI*^{time} and *SpecificationII* contain a set of user variables for each user and a set of channel variables for each channel, the specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} contain only a single set of channel variables and two sets of user variables because these specifications model only two users and a single payment channel. The figure indicates how the mappings between the specifications change the values of the variables: The mapping m_c removes values that are unrelated to channel c from global variables and user variables but leaves the values of channel variables unchanged. The refinement mapping f leaves user variables and global variables unchanged but changes the values of channel variables because in *SpecificationII*^{single} payment channels are modeled more abstractly than in *SpecificationI*$^{time,\ single}$. The mapping g leaves global variables and user variables unchanged and performs the same changes to channel variables as the refinement mapping f. While we do not explicitly specify a mapping from *SpecificationII*^{single} to *SpecificationII*, the changes by the mapping g imply a mapping from *SpecificationII*^{single} to *SpecificationII* that leaves channel variables unchanged and merges the values of different channels in global variables and user variables (reversing the changes of the mapping m_c).

SpecificationII^{time} have variables for all users and channels in a payment channel network, the single-channel specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} have only those variables that are specific to the two users and the channel that is modeled by a single-channel specification. Technically, this means for example that, while *UserHonest* maps in *SpecificationI*^{time} each user $u \in U$ to a variable $UserHonest[u]$, in *SpecificationI*$^{time,\ single}$, the domain of *UserHonest* contains only the two users modeled by *SpecificationI*$^{time,\ single}$.

The single-channel specifications *SpecificationI*$^{time,\ single}$ and *SpecificationII*^{single} are meant to model a user's behavior in a setting where the other channels in the network are modeled by the module *ChannelEnvironment*. For this purpose, the single-channel specifications contain the additional variables *UserRequestedInvoices[u]* for each user u of the modeled channel. These variables store state of the environment. Specifically, these variables are sets that contain payments for which a user in the environment has requested an invoice.

This set is required to ensure that at most one invoice is requested for each payment. This prevents the existence of behaviors of infinite length in which the environment requests an invoice over and over again.

The single-channel specifications include the additional module *ChannelEnvironment* to model the environment of a channel. We briefly present the seven actions specified in the module *ChannelEnvironment*. By the phrase 'a user in the environment' we refer to a user $u \in U$ that is not one of the two users of the channel modeled by the single-channel specification, i.e. $u \notin U_c$. The first four actions describe steps of a user in the environment. The later three actions model steps of one of the users of the modeled payment channel c that are steps of an instance of the module *HTLCUser* for another channel than channel c.

- The action *RequestInvoice* models that a user in the environment requests an invoice from one of the users of channel c. The variable $UserRequestedInvoices[u]$ of the user u from which the invoice was requested is used to prevent that the same invoice is requested multiple times.
- The action *GenerateAndSendPaymentHash* models that a user in the environment from whom an invoice was requested replies with an `invoice` message that contains a hash and a payment secret.
- The action *IgnoreMessageDuringClosing* models that a user in the environment ignores a request for an invoice.
- The action *ReceiveInvoice* models that a user in the environment who has requested an invoice reads and removes the invoice from the variable *Messages*.
- The action *AddOutgoingHTLCToOtherChannel* models that a user u who is one of the users of the modeled channel c adds a payment that is contained in the set $UserNewPayments[u]$ to another channel than the channel c.
- The action *AddNewForwardedPayment* models that a user u who is one of the users of the modeled channel c adds a new payment to $UserNewPayments[u]$ because an incoming HTLC was received on another channel then the channel c.
- The action *ReceivePreimageForHTLC* models that user receives a preimage for an HTLC that is part of the modeled channel on another channel than the modeled channel. This would typically be a channel with the corresponding outgoing HTLC.

Table 4.1 shows how steps of actions of the modules *PaymentChannelUser* and *HTLCUser* are mapped to steps of actions of the module *ChannelEnvironment*.

An instance of $SpecificationI^{time,\ single}$ defines the behavior of one single channel. To define an instance of $SpecificationI^{time,\ single}$ for each channel $c \in C$, we use the function m_c to define a state of $SpecificationI^{time,\ single}$ based on a state of $SpecificationI^{time}$.

The mapping m_c is formally defined in TLA$^+$ in the appendix in Fig. A.1. Thus, the following Definition 19 does not aim for being perfectly precise but aims to give an understanding of the definition of m_c. Also, the following definition is not complete in the sense that it does not define the values of all variables but contains only an exemplary

Step of *SpecificationI*^{time}	Step of *SpecificationI*$^{time,\ single}$ for channel B
PCU(c, u)!NoteThatHTLCFulfilledOnChain	CEnv(c', u)!ReceivePreimageForIncomingHTLC
HU(c, u)!ReceiveHTLCPreimage	CEnv(c', u)!ReceivePreimageForIncomingHTLC
HU(c, u)!RequestInvoice	CEnv(c', u)!RequestInvoice
HU(c, u)!GenerateAndSendPaymentHash	CEnv(c', u)!GenerateAndSendPaymentHash
HU(c, u)!ReceivePaymentHash	CEnv(c', u)!ReceivePaymentHash
HU(c, u)!AddAndSendOutgoingHTLC	CEnv(c', u)!AddOutgoingHTLCToOtherChannel
HU(c, u)!ReceiveUpdateAddHTLC	CEnv(c', u)!AddNewForwardedPayment

Table 4.1.: Mapping of steps of user u in channel c in *SpecificationI*^{time} to steps of channel $c' \neq c$ of user u in *SpecificationI*$^{time,\ single}$ if the step in *SpecificationI*$^{time,\ single}$ is not a stuttering step.

selection of variables. As indicated in Fig. 4.27, the mapping m_c selects the variables of channel c, the variables of the users of channel c, and the global variables from a state of *SpecificationI*^{time} and creates a state of *SpecificationI*$^{time,\ single}$ with those variables. Because the user variables and the global variables might contain values that are irrelevant for channel c, the mapping m_c reduces the values of these variables. For example, if the user variable $UserPreimageInventory[u]$ in the state of *SpecificationI*^{time} contains a preimage for an HTLC that does not exist and will not exist in channel c, then this preimage will not be contained in the value of $UserPreimageInventory[u]$ in the state of *SpecificationI*$^{time,\ single}$.

Definition 19 (Mapping m_c of channel c in a state of *SpecificationI*^{time} to a state of *SpecificationI*$^{time,\ single}$)**.** Define a mapping m_c from the state space of *SpecificationI*^{time} to the state space of *SpecificationI*$^{time,\ single}$ so that for a state s of *SpecificationI*^{time}, the state $m_c(s)$ assigns to all variables of *SpecificationI*$^{time,\ single}$ the following values:

- Channel variables and ⟨channel and user⟩ variables are unchanged: Each variable of the channel variables of channel c and the ⟨channel and user⟩ variables of user u and channel c in *SpecificationI*$^{time,\ single}$ is assigned the value of the corresponding variable in state s.
- The user variables in *SpecificationI*$^{time,\ single}$ are reduced to the values that are relevant for the channel c. The user variables are assigned the following values for the two users $u \in U_c$.
  - *UserPreimageInventory[u]* is assigned the value of s.*UserPreimageInventory[u]*$\cap$ $R_{u,c}$ where $R_{u,c}$ is the set of relevant preimages for each user u of the channel c. The preimages that are relevant for user u in channel c are the preimages for payments received by user u and the preimages for all HTLCs forwarded by user u (formally defined in TLA$^+$ in the appendix in Fig. A.3).
  - *UserPayments[u]* is assigned the set of all payments in s.*UserPayments[u]* that are received or sent by user u over channel c.
  - *UserHonest[u]* is assigned the corresponding value in state s.

- The global variables in *SpecificationI*$^{time,\ single}$ are are also reduced to the values that are relevant for channel c. These variables are assigned the following values:
  - *Messages* is assigned a set that contains all messages in $s.$*Messages* for which either the sender or the recipient is a user who is a user of channel c.
  - *LedgerTx* is the set of all transactions in $s.$*LedgerTx* that are related to channel c. These are the transaction that contains the funding output and any transaction directly or indirectly spending the funding output.
  - *TxAge* is assigned the function in $s.$*TxAge* with the function's domain restricted to the id's of transactions that are in *LedgerTx*, i.e. transactions that are related to channel c.
  - *LedgerTime* is assigned the maximum of the value that was assigned to *LedgerTime* for channel c in the previous step[7] and the set of values that are in the set of relaxed zone time points $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ and are lower than or equal to $s.LedgerTime$.

To explain the reasons behind the definition of the mapping m_c, we briefly discuss the mapping of each variable mentioned in Definition 19. The values of the variables that are specific to a channel (channel variables and ⟨channel and user⟩ variables) are unchanged by the mapping m_c because these variables occur only in the instance of *SpecificationI*$^{time,\ single}$ for channel c but not in the instance of *SpecificationI*$^{time,\ single}$ for any other channel $c' \neq c$.

The value of the variable *UserPreimageInventory[u]* is reduced to the preimages that are relevant for channel c. If user u learns a preimage for which no HTLC in channel c exists in the current state or in any successor state, the knowledge of the preimage does not affect the actions related to channel c and, thus, this change should not be visible for an instance of *SpecificationI*$^{time,\ single}$ for channel c. Analogously, from the value of the variable *UserPayments[u]*, payments are removed that are not routed over channel c because these payments are irrelevant for channel c. The variable *UserHonest[u]* is an example of a user variable whose value is unchanged by the mapping m_c.

The global variable *Messages* in *SpecificationI*^{time} might contain messages that are exchanged between any pair of modeled users. For channel c, messages exchanged between users who are not part of channel c are irrelevant and should not be modeled in an instance of *SpecificationI*$^{time,\ single}$ for channel c. Thus, the mapping m_c removes such messages from the variable *Messages*. Similarly, the values of the variables *LedgerTx* and *TxAge* should contain only transactions that are related to channel c. Modeling changes in the blockchain that are irrelevant for channel c would unnecessarily complicate the specification of *SpecificationI*$^{time,\ single}$ and increase the specification's state space. Thus, the mapping m_c selects only those transactions that are relevant for channel c. Finally, the

[7] This value is obtained by adding a helper variable for each channel c to *SpecificationI*^{time} that stores the mapped value of *LedgerTime* for channel c.

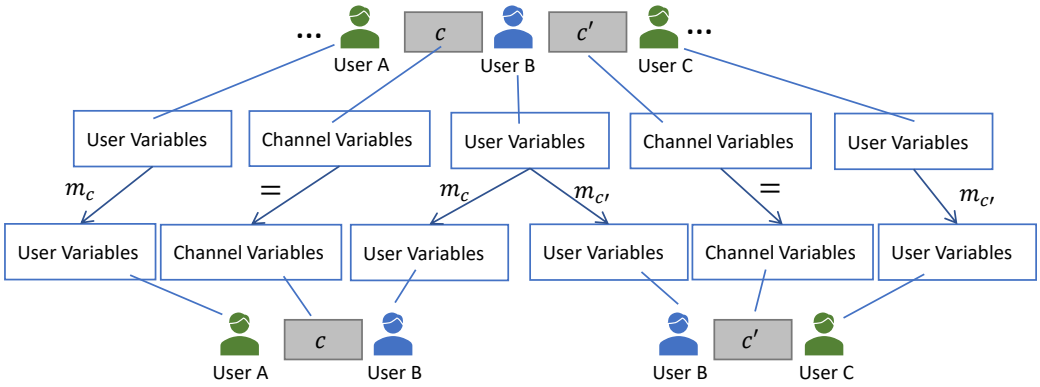

Figure 4.28.: Sketch showing how the mapping m_c maps the variables of *SpecificationI*^{time} (top) with two channels c and c' to the variables of two instances of *SpecificationI*$^{time,\ single}$ (bottom) for the channels c and c'. Global variables are handled in this context like user variables and are not shown for simplicity. Similarly, ⟨channel and user⟩ variables are handled like channel variables. The values of user variables are reduced by the mapping m_c to values that are relevant for channel c. Channel variables are unchanged by the mapping m_c. If a user has multiple payment channels (see user B), the user's user variables are mapped into multiple instances of *SpecificationI*$^{time,\ single}$. However, the values of the user variables in the two instances of *SpecificationI*$^{time,\ single}$ for channel c and c' might differ because the channel specific mapping is used (m_c and $m_{c'}$).

value of the variable *LedgerTime* must be adjusted by the mapping m_c because *SpecificationI*$^{time,\ single}$ is, as well as *SpecificationI*^{time}, a time-optimized specification in which time advances only to zone time points (see Section 4.1). The set of zone time points in *SpecificationI*^{time} can be seen as a composition of the sets of zone time points of each channel c. Thus, a clock value that is a zone time point in *SpecificationI*^{time} might not be a zone time point in *SpecificationI*$^{time,\ single}$ for a specific channel c. Because the value of *LedgerTime* must be a zone time point in *SpecificationI*$^{time,\ single}$, the mapping m_c maps the value of *LedgerTime* in *SpecificationI*^{time} to the corresponding zone time point of *SpecificationI*$^{time,\ single}$. This is the greatest zone time point that is lower than or equal to the value of *LedgerTime* in *SpecificationI*^{time}. However, to prevent that the value of *LedgerTime* in *SpecificationI*$^{time,\ single}$ decreases during a step in which a zone time point is removed, the value of *LedgerTime* in *SpecificationI*$^{time,\ single}$ is set to at least the variable's value in the previous state.

We prove that the definition of the mapping m_c maps each behavior of *SpecificationI*^{time} to a behavior of *SpecificationI*$^{time,\ single}$ in Lemma 8. Before that, we define Lemma 7 which is a special case for the proof of Lemma 8. Lemma 7 states that the module *ChannelEnvironment* abstracts the effects on the payment channel c of all steps that are taken in other payment channels. We will use Lemma 7 to prove Lemma 8.

More specifically, the idea behind Lemma 7 is as follows: Each channel in *SpecificationI*^{time} is mapped to an instance of *SpecificationI*$^{time,\ single}$. While the channel variables for each channel c can be only in the instance of *SpecificationI*$^{time,\ single}$ for channel c, two channels c and c' of the same user u both share the same user variables for user u (see Fig. 4.28). While the mapping m_c reduces these user variables by removing values in the variables that are not relevant for the channel c, there are values of user variables that are rele-

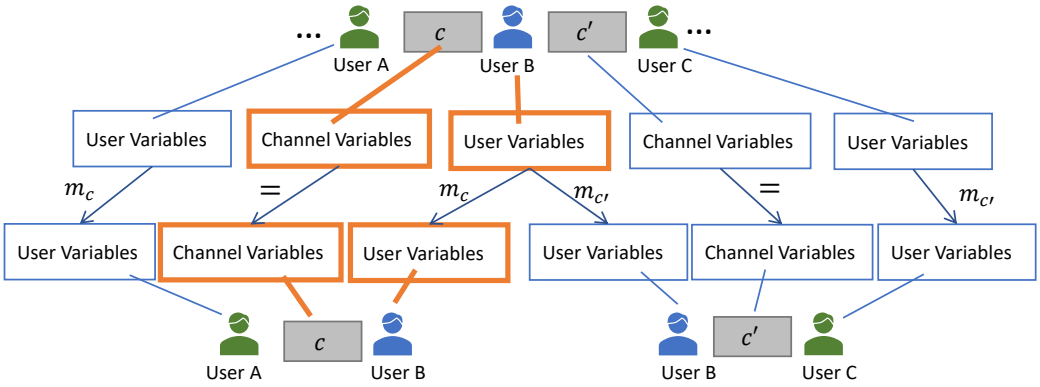

Figure 4.29.: Variation of Fig. 4.28 with the variables highlighted that are changed during a step of user B in channel c that changes only channel variables and user variables whose values are only relevant for the instance of $SpecificationI^{time,\ single}$ for channel c.

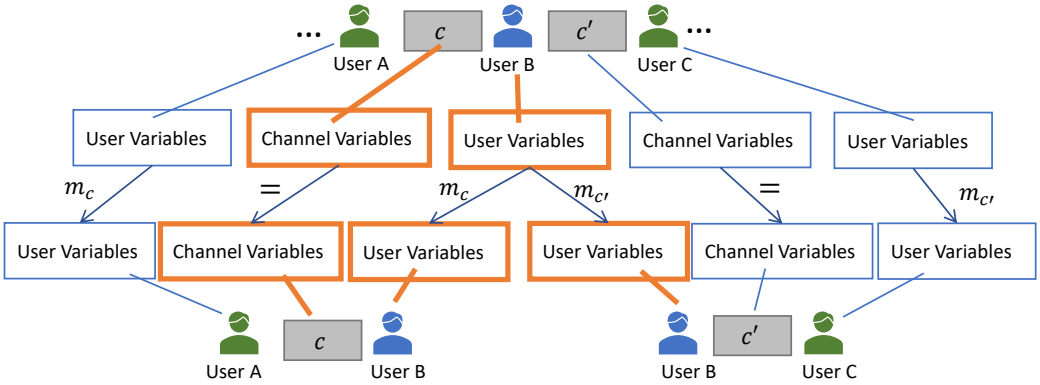

Figure 4.30.: Variation of Fig. 4.28 with the variables highlighted that are changed during a step of user B in channel c that changes channel variables and user variables whose values are relevant for the instance of $SpecificationI^{time,\ single}$ for channel c and for channel c'. Thus, the step in $SpecificationI^{time}$ implies a step in both instances of $SpecificationI^{time,\ single}$. An example of such a step is step in channel c in that user B retrieves a preimage that is also relevant for channel c'.

vant for two channels. This can, for example, be a preimage for an HTLC that exists in two channels because user u forwards the HTLC for a multi-hop payment. Most steps in $SpecificationI^{time}$ by user u in channel c will update the user variables of user u and the channel variables of channel c and, if the steps are mapped to $SpecificationI^{time,\ single}$, these steps will only update the user variables and channel variables in the instance of $SpecificationI^{time,\ single}$ for channel c (see Fig. 4.29). However, because some user variables are shared between channels, it might happen that a step of user u in channel c also leads to changed user variables in another channel $c' \neq c$ (see Fig. 4.30). To ensure that each behavior of $SpecificationI^{time}$ is mapped to a behavior of $SpecificationI^{time,\ single}$ for each channel, $SpecificationI^{time,\ single}$ must explicitly allow such changes to the variables of $SpecificationI^{time,\ single}$. Because, from the perspective of a specific channel, such changes come from the environment of the channel, we describe these steps in the module *ChannelEnvironment*. Lemma 7 shows that every step in $SpecificationI^{time}$ in a channel c is mapped to a step of $SpecificationI^{time,\ single}$ in channel $c' \neq c$. The step can be either a

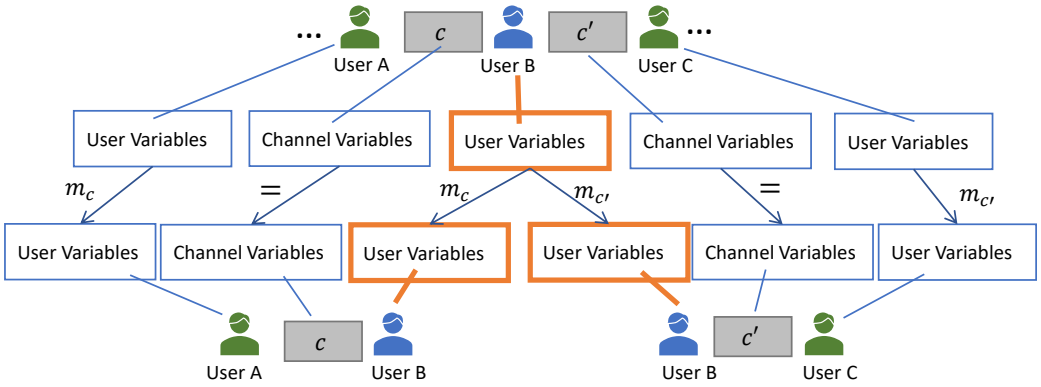

Figure 4.31.: Variation of Fig. 4.28 with the variables highlighted that are changed during a step of user B in channel c that changes user variables whose values are relevant for the instance of *SpecificationI*$^{time,\ single}$ for channel c and for channel c'. Thus, the step in *SpecificationI*^{time} implies a step in both instances of *SpecificationI*$^{time,\ single}$. An example of such a step is a step of the module *HTLCUser* for requesting an invoice because the corresponding action is independent of the specific channel.

stuttering step (if the variables of channel c' are unchanged, see Fig. 4.29), a step of the module *ChannelEnvironment*, or a step of *HTLCUser*. The step can be a step of *HTLCUser* because there are three actions in the module *HTLCUser* that are not channel-specific (*RequestInvoice*, *GenerateAndSendPaymentHash*, and *ReceivePaymentHash*). A step of each of these three actions by user u in channel c changes only the user variables of user u (see Fig. 4.31) and, thus, is a valid step of *HTLCUser* in *SpecificationI*$^{time,\ single}$ for all channels that user u is part of.

Lemma 7. *Given a state s of SpecificationI^{time}, it holds that for every step $s \rightarrow t$ of an action of PaymentChannelUser or HTLCUser for channel c, in all channels $c' \neq c$ the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an instance of ChannelEnvironment for channel c' or a or a step of HTLCUser for channel c' and the same user as in channel c.*

We prove Lemma 7 in the appendix in Section A.3.10 and sketch the proof here. To prove Lemma 7, we assume that a step $s \rightarrow t$ is taken in *SpecificationI*^{time} in channel c. We discuss all user variables and global variables that might be changed in a channel $c' \neq c$ in the step $m_{c'}(s) \rightarrow m_{c'}(t)$. We show that the global variables *LedgerTime*, *LedgerTx*, and *TxAge* are always unchanged in step $m_{c'}(s) \rightarrow m_{c'}(t)$ because, even if these variables change in the step $s \rightarrow t$ of *SpecificationI*^{time}, all possible changes are hidden by the mapping $m_{c'}$. For the global variable *Messages* and the user variables, we determine all actions of the module *PaymentChannelUser* and *HTLCUser* that change the value of these variables in *SpecificationI*^{time}. Then, we show individually for each action A of these eleven actions that, if the step $s \rightarrow t$ is an A step, the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of *ChannelEnvironment*, or a step of *HTLCUser*. For the three actions of *HTLCUser* that are not channel specific, we show that a step $s \rightarrow t$ of one of these actions of user u in *SpecificationI*^{time} is mapped to a step $m_{c'}(s) \rightarrow m_{c'}(t)$ of the same action in *SpecificationI*$^{time,\ single}$ for channel c' if user u is part of channel c'. Otherwise, the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step. For each of the remaining actions, we show

in which cases the step $s \rightarrow t$ leads to the step $m_{c'}(s) \rightarrow m_{c'}(t)$ not being a stuttering step. Then, we show that under these circumstances the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is a step of an action of *ChannelEnvironment* in *SpecificationI*$^{time,\ single}$. □

We additionally check the correctness of the proof of Lemma 7 using random simulation which is an approach for partial model checking as presented in Section 4.7.2.

Having shown that for a step in channel c the mapping m_c maps the state of all other channels $c' \neq c$ correctly to *SpecificationI*$^{time,\ single}$, we now show that the mapping m_c also maps each step in *SpecificationI*^{time} to a step in *SpecificationI*$^{time,\ single}$ for channel c.

Lemma 8. *For each channel $c \in C$, the mapping m_c maps each behavior $\sigma = \langle \sigma_0, \sigma_1, \sigma_2, ... \rangle$ of SpecificationI^{time} to a behavior $\langle m_c(\sigma_0), m_c(\sigma_1), m_c(\sigma_2), ... \rangle$ of SpecificationI$^{time,\ single}$ of channel c.*

We prove Lemma 8 in the appendix in Section A.3.11 and sketch the ideas of the proof here. We prove that every behavior of *SpecificationI*^{time} is mapped to a behavior of *SpecificationI*$^{time,\ single}$ by proving that each initial state of *SpecificationI*^{time} is mapped to an initial state of *SpecificationI*$^{time,\ single}$ and that each step of *SpecificationI*^{time} is mapped to a step of *SpecificationI*$^{time,\ single}$.

The first statement that each initial state s of *SpecificationI*^{time} is mapped to an initial state $m_c(s)$ of *SpecificationI*$^{time,\ single}$, follows directly from the definitions of the mapping m_c and the $Init$ predicates of the specifications. To show the second statement that each step $s \rightarrow t$ of *SpecificationI*^{time} is mapped to a step $m_c(s) \rightarrow m_c(t)$ of *SpecificationI*$^{time,\ single}$, we distinguish the cases that the step $s \rightarrow t$ is a step of the module *PaymentChannelUser*, the module *HTLCUser*, the module *LedgerTime*, or a final withdraw step.

- Consider the case that the step $s \rightarrow t$ is a step of the module *PaymentChannelUser* or *HTLCUser* in *SpecificationI*^{time}. We distinguish the cases that the step $s \rightarrow t$ is a step for channel c or a step for channel $c' \neq c$.

  Consider the first case that the step $s \rightarrow t$ is a step for channel c. Let A be the action of the module *PaymentChannelUser* or *HTLCUser* that describes the step $s \rightarrow t$. The definition of the action A is a conjunction of several conditions that need to be fulfilled. From the assumption that the step $s \rightarrow t$ is an A step, we know that the conditions of action A are fulfilled for the step $s \rightarrow t$. Based on this, we prove that the same conditions are fulfilled for the step $m_c(s) \rightarrow m_c(t)$. There are basically two different types of cases depending on whether the variables used in a condition are changed by the mapping m_c or not. We give examples for both types. The first example is a representative for all channel specific variables that are unchanged by the mapping m_c. The second example is an example of those user variables and global variables that are changed by the mapping m_c.

Example 4.11. Consider the condition $ChannelUserState[c][u]$ = "init" which is a condition of the action *SendOpenChannel* of the module *PaymentChannelUser*. Because the step $s \rightarrow t$ is a step for channel c, the variable *ChannelUserState[c][u]* is the same variable in the steps $s \rightarrow t$ and $m_c(s) \rightarrow m_c(t)$. Because the variable *ChannelUserState* is unchanged by the mapping m_c, it directly follows that the condition $ChannelUserState[c][u]$ = "init" is valid in the step $m_c(s) \rightarrow m_c(t)$.

Example 4.12. Consider the condition

$$UserPreimageInventory' = [\,UserPreimageInventory \text{ EXCEPT } ![u] = @ \cup \{message.data.preimage\}]$$

which is a condition of the action *ReceiveHTLCPreimage* of the module *HTLCUser*. The action receives a message that contains the preimage of an HTLC in the channel. The mapping m_c reduces the value of the variable *UserPreimageInventory[u]* to those preimages that are relevant to channel c which includes preimages of HTLCs of the channel. Thus, if the value *message.data.preimage* is contained in the set *UserPreimageInventory[u]* in the step $s \rightarrow t$ then the value is also contained in the set *UserPreimageInventory[u]* in step $m_c(s) \rightarrow m_c(t)$.

In the proof, we discuss all actions of the modules *PaymentChannelUser* and *HTLCUser* and show that, if the step $s \rightarrow t$ is a step of action A, then the step $m_c(s) \rightarrow m_c(t)$ is also a step of action A.

Consider the second case that the step $s \rightarrow t$ is a step for channel $c' \neq c$. It follows from Lemma 7 that the step $m_c(s) \rightarrow m_c(t)$ is a stuttering step, a step of an action of the module *ChannelEnvironment*, or a step of the module *HTLCUser*.

- Consider the case that the step $s \rightarrow t$ is a step of the module *LedgerTime* in *SpecificationI*^{time}. The module *LedgerTime* has a single action that advances one or multiple clocks. We prove that the step $m_c(s) \rightarrow m_c(t)$ is a step of the module *LedgerTime* in *SpecificationI*$^{time,\ single}$ by showing that all conditions of the action of the module *LedgerTime* are fulfilled. Specifically, this means that we show that the values of the clocks do not decrease, time bounds are respected, and all non-clock variables are unchanged.
- Finally, consider the case that the step $s \rightarrow t$ is a step of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationI*^{time}. We show that the step $m_c(s) \rightarrow m_c(t)$ is a step of the action *WithdrawBalancesAfterChannelClosed* analogously to the first case and first subcase above, i.e., we show that each condition of the action *WithdrawBalancesAfterChannelClosed* in step $s \rightarrow t$ implies that the same condition is fulfilled in the step $m_c(s) \rightarrow m_c(t)$ where some user variables and global variables are mapped to reduced values.

Thereby, we have shown that *SpecificationI*$^{time,\ single}$ describes the behavior of any payment channel that is modeled in *SpecificationI*^{time}. □

	Step of $SpecificationI^{time,\ single}$	Step of $SpecificationII^{single}$
1	PCU(1,1)!SendOpenChannel	
2	PCU(1,2)!SendAcceptChannel	
3	PCU(1,1)!CreateFundingTransaction	
4	HU(1,1)!RequestInvoice	HU(1,1)!RequestInvoice
5	HU(1,2)!GenerateAndSendPaymentHash	HU(1,2)!GenerateAndSendPaymentHash
6	PCU(1,1)!SendSignedFirstCommitTransaction	
7	PCU(1,2)!ReplyWithFirstCommitTransaction	
8	PCU(1,1)!ReceiveCommitTransaction	
9	PCU(1,1)!PublishFundingTransaction	AE(1)!OpenPaymentChannel
10	PCU(1,1)!SendNewRevocationKey	
11	PCU(1,2)!NoteThatFundingTransactionPublished	
12	PCU(1,2)!SendNewRevocationKey	
13	PCU(1,1)!ReceiveNewRevocationKey	
14	PCU(1,2)!ReceiveNewRevocationKey	
15	HU(1,1)!ReceivePaymentHash	HU(1,1)!ReceivePaymentHash
16	HU(1,1)!AddAndSendOutgoingHTLC	HU(1,1)!AddAndSendOutgoingHTLC
17	PCU(1,1)!SendSignedCommitment	
18	HU(1,2)!ReceiveUpdateAddHTLC	HU(1,2)!ReceiveUpdateAddHTLC
19	PCU(1,2)!ReceiveSignedCommitment	
20	PCU(1,2)!CloseChannel	AE(1)!SetOnChainHTLCsAndCheater
21	PCU(1,1)!NoteThatOtherUserClosedHonestly	AE(1)!CommitHTLCsOnChain
22	PCU(1,2)!ResolveHTLCAfterClose	AE(1)!FulfillIncomingHTLCsOnChain
23	PCU(1,1)!NoteThatHTLCFulfilledOnChain	AE(1)!NoteFulfilledHTLCsOnChain
24	PCU(1,1)!Terminate	
25	PCU(1,2)!Terminate	AE(1)!ClosePaymentChannel

Table 4.2.: Trace of an execution in which a channel is opened and a payment is made that is resolved on the blockchain. The left column shows the steps of $SpecificationI^{time,\ single}$ while the right column shows the matching steps of $SpecificationII^{single}$. In this trace, there is one channel with id 1 (first parameter of PCU, HU and AE) and there are two users with ids 1 and 2 (second parameter of PCU and HU). $SpecificationII^{single}$ contains the module *AbstractEnforcement* and the module *HTLCUser* parameterized for each user. The right column shows which action of these modules describes each step. For better readability, we omit stuttering steps; if the right column is empty, the step of $SpecificationI^{time,\ single}$ is mapped to a stuttering step in $SpecificationII^{single}$.

4.3.3.2. $SpecificationI^{time,\ single}$ Implements $SpecificationII^{single}$

Having defined $SpecificationI^{time,\ single}$ and $SpecificationII^{single}$ in the previous subsection, we now check that $SpecificationI^{time,\ single}$ implements $SpecificationII^{single}$. To this end, we define a mapping f and use model checking to check that the mapping f is a refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$.

Definition 20 (Refinement mapping f from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$)**.** Define f to be the refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$. The refinement mapping f is defined in TLA$^+$ in the module *SpecificationItimetoII* in [GH26b].

The refinement mapping f maps a state of $SpecificationI^{time,\ single}$ to a state of $SpecificationII^{single}$. We explain the idea behind this mapping with an example trace shown in Table 4.2. In the appendix, we give a further example of a behavior in which a channel is opened and an off-chain update is successfully completed (Table A.14 on page 333).

Example 4.13. This trace shows a behavior of $SpecificationI^{time,\ single}$ in the left column and the corresponding behavior of $SpecificationII^{single}$ in the right column. For each step in the behaviors, the corresponding step in the other behavior is shown in the same line. In this example, we refer to any step of $SpecificationI^{time,\ single}$ as a step $s \rightarrow t$ and to the corresponding step in $SpecificationII^{single}$ to which this step is mapped by the refinement mapping f as the step $f(s) \rightarrow f(t)$. In the behavior shown in Table 4.2, a channel is opened, a payment is added, an update of the channel is started, the channel is closed by committing an HTLC on the blockchain, and the HTLC is fulfilled on the blockchain.

For lines in which the right column contains no entry, there is a stuttering step in $SpecificationII^{single}$. However, a step $s \rightarrow t$ in the same line as a stuttering step updates variables that exist also in $SpecificationII^{single}$. Thus, the mapping f must be constructed so that the step $f(s) \rightarrow f(t)$ is a stuttering step by 'hiding' the changes that happen in step $s \rightarrow t$. Such a 'hiding' is implemented by defining the mapping f so that the value of each variable in state $f(t)$ equals the value of the same variable in state $f(s)$.

A challenge for defining the mapping f is that a series of multiple step of $SpecificationI^{time,\ single}$ (e.g., lines 1, 2, 3, 6, 7, 8 in Table 4.2) must be mapped to stuttering steps until the step in line 9 in which the funding transaction is published and the corresponding step in $SpecificationII^{single}$ is the step in which the channel is opened. Note that such a series of steps of the module *PaymentChannelUser* that is mapped to a single step in the module *AbstractEnforcement* might be interleaved with steps of the module *HTLCUser* (e.g., lines 4, 5 in Table 4.2). This shows that the mapping f does not only replace a series of steps of the module *PaymentChannelUser* by a single step of the module *AbstractEnforcement* but the mapping must also ensure that interleaved steps of other modules can be reordered before the series of steps that is replaced. To achieve this property, the mapping f must not change values of variables that have an effect on the actions of other modules.

A further challenge is that an off-chain update of a payment channel might be started in $SpecificationI^{time,\ single}$ (lines 17 and 19 of Table 4.2) but the off-chain update is not completed but the channel is closed (line 20) and the existing HTLC is fulfilled on-chain.

To achieve that the mapping f does not change values of variables that other modules depend on, the mapping f maps user variables and global variables in $SpecificationII^{single}$ to the variables' values in $SpecificationI^{time,\ single}$. Only the values of the channel variables *ChannelMessages*, *ChannelPendingBalance*, *ChannelUserState*, *ChannelUserBalance*, and *ChannelUserVars* are changed. In the following, we explain how the mapping f changes the values of these channel variables.

The variable *ChannelMessages* is a set that contains the in-flight messages that are exchanged between the users of the channel. The mapping f removes all messages from the variable *ChannelMessages* that are sent or received by the module *PaymentChannelUser* so that only messages that are sent by the module *HTLCUser* are left.

The variable *ChannelUserState* is mapped to a value depending on the current state of the payment channel: While the channel is being opened, the current value is mapped to `init`. When the channel is open, while the channel is being updated, and while a closing of the channel is being started, the channel's state is mapped to `idle`. If a commitment transaction has been published, the state is mapped to `closing`.

The variable *ChannelUserVars* has the most complicated mapping. While a channel is being opened or updated and while the channel is in the process of being closed, the states of HTLCs are updated. The mapping f hides these changes in the abstraction by 'rolling back' the changes to last state at which a step in the module *AbstractEnforcement* was taken. For example, when an HTLC is added, the HTLC has the intermediate states SENT-COMMIT, RECV-COMMIT, and PENDING-COMMIT in the module *PaymentChannelUser* while in the module *AbstractEnforcement* an HTLC's state is updated directly from NEW to COMMITTED (see Figs. 3.13 and 4.25). Because the module *AbstractEnforcement* updates the variables of both users in a single step, the mapping f has to consider both users' variables and, for example, if an HTLC is already in the state COMMITTED but not yet in the other user's state, the HTLC's state in both user's variables is mapped to NEW. A challenge for defining the mapping f is that updating and closing a channel can interleave and during closing the state of an HTLC can be changed by off-chain messages as well as by on-chain transactions which leads to a vast set of behaviors and edge cases that must be considered.

The variables *ChannelPendingBalance* and *ChannelUserBalance* are mapped to values matching the values of the mapped HTLC states in the variable *ChannelUserVars*.

Example 4.14. Table 4.3 shows a behavior in which a channel is opened and an update of the channel to add an HTLC is started. Just before the update is completed, the channel is closed and the HTLC is fulfilled on the blockchain. Table 4.3 also shows the state of the HTLC for each user in $\mathit{SpecificationI}^{\mathit{time},\ \mathit{single}}$ and the state of the HTLC in each corresponding state of $\mathit{SpecificationII}^{\mathit{single}}$.

The example shows a behavior that is challenging to map by the refinement mapping f. We briefly discuss why this is the case. The example shows that an HTLC's state is mapped to NEW until the HTLC is committed. However, even if the HTLC is committed for one user, the mapping f must map the HTLC's state to NEW until the HTLC is committed for both users because the module *AbstractEnforcement* updates the state of the HTLC for both users at once. In the specific case shown in Table 4.3, the HTLC is started to be committed on-chain but then the channel is closed by committing the HTLC on-chain. In this case, in $\mathit{SpecificationII}^{\mathit{single}}$, the HTLC is never committed off-chain but only committed on-chain. However, because the HTLC is fulfilled on-chain by one user (line 26) before the other user has even noticed that the channel has been closed (line 27), the HTLC's state of user 2 changes directly from NEW to PERSISTED. The HTLC's state is advanced to COMMITTED for user 1 when user 1 notices that the channels has been closed and the HTLC has been committed on-chain (line 27).

	Step of $SpecificationI^{time,\ single}$	Step of $SpecificationII^{single}$	HTLC State in Protocol User 1	HTLC State in Abstract User 1	HTLC State in Protocol User 2	HTLC State in Abstract User 2
1	PCU(1)!SendOpenChannel					
2	PCU(2)!SendAcceptChannel					
3	PCU(1)!CreateFundingTransaction					
4	PCU(1)!SendSignedFirstCommitTransaction					
5	PCU(2)!ReplyWithFirstCommitTransaction					
6	PCU(1)!ReceiveCommitTransaction					
7	PCU(1)!PublishFundingTransaction	AE!OpenPaymentChannel				
8	PCU(2)!NoteThatFundingTransactionPublished					
9	PCU(1)!SendNewRevocationKey					
10	PCU(2)!SendNewRevocationKey					
11	PCU(2)!ReceiveNewRevocationKey					
12	PCU(1)!ReceiveNewRevocationKey					
13	HU(1)!RequestInvoice	HU(1)!RequestInvoice				
14	HU(2)!GenerateAndSendPaymentHash	HU(2)!GenerateAndSendPaymentHash				
15	HU(1)!ReceivePaymentHash	HU(1)!ReceivePaymentHash				
16	HU(1)!AddAndSendOutgoingHTLC	HU(1)!AddAndSendOutgoingHTLC	NEW	NEW		
17	HU(2)!ReceiveUpdateAddHTLC	HU(2)!ReceiveUpdateAddHTLC	NEW	NEW	NEW	NEW
18	PCU(1)!SendSignedCommitment		SENT-COMMIT	NEW	NEW	NEW
19	PCU(2)!ReceiveSignedCommitment		SENT-COMMIT	NEW	RECV-COMMIT	NEW
20	PCU(2)!RevokeAndAck		SENT-COMMIT	NEW	PENDING-COMMIT	NEW
21	PCU(1)!ReceiveRevocationKey		PENDING-COMMIT	NEW	PENDING-COMMIT	NEW
22	PCU(2)!SendSignedCommitment		PENDING-COMMIT	NEW	SENT-COMMIT	NEW
23	PCU(1)!ReceiveSignedCommitment		RECV-COMMIT	NEW	SENT-COMMIT	NEW
24	PCU(1)!RevokeAndAck		COMMITTED	NEW	SENT-COMMIT	NEW
25	PCU(2)!CloseChannel	AE!SetOnChainHTLCsAndCheater	COMMITTED	NEW	COMMITTED	NEW
26	PCU(2)!ResolveHTLCAfterClose	AE!FulfillIncomingHTLCsOnChain	COMMITTED	NEW	PERSISTED	PERSISTED
27	PCU(1)!NoteThatOtherUserClosedHonestly	AE!CommitHTLCsOnChain	COMMITTED	COMMITTED	PERSISTED	PERSISTED
28	PCU(1)!NoteThatHTLCFulfilledOnChain	AE!NoteFulfilledHTLCsOnChain	PERSISTED	PERSISTED	PERSISTED	PERSISTED
29	PCU(1)!Terminate		PERSISTED	PERSISTED	PERSISTED	PERSISTED
30	PCU(2)!ReceiveRevocationKey		PERSISTED	PERSISTED	PERSISTED	PERSISTED
31	PCU(2)!Terminate	AE!ClosePaymentChannel	PERSISTED	PERSISTED	PERSISTED	PERSISTED

Table 4.3.: Trace of an execution in which a channel is opened and a payment is made from user 1 to user 2. Each step starts with a short identifier of the module and the user that performs the step. The trace shows an example of a channel update overlapping with the closing of a channel. The channel is opened in step 7, an HTLC is created in step 16.The update of the channel to commit the HTLC starts in step 18. In step 25, the channel is closed by user 2 before the update is complete. The published commitment transaction contains the HTLC and, thus, user 2 advances the HTLC's state to COMMITTED. The receiver of the payment (user 2) fulfills the HTLC on the blockchain in step 26. In step 27, user 1 notices that the channel has been closed and advances the HTLC's state to COMMITTED.

Having defined the mapping f, we verify that the mapping f is a refinement mapping from *SpecificationI*$^{time,\ single}$ to *SpecificationII*^{single}. To this end, we model check Lemma 9:

Lemma 9. *SpecificationI*$^{time,\ single}$ *for channel* $c \in C$ *is a refinement of SpecificationII*^{single} *with the refinement mapping* f*.*

We check the correctness of this lemma using model checking (see Section 4.7). We model check *SpecificationI*$^{time,\ single}$ and verify that applying the mapping f to each step $s \rightarrow t$ of *SpecificationI*$^{time,\ single}$ leads to steps $f(s) \rightarrow f(t)$ of *SpecificationII*^{single} and that each behavior of *SpecificationI*$^{time,\ single}$ is mapped to a behavior of *SpecificationII*^{single}.

4.3.3.3. Composition of *SpecificationII*^{single} Implements *SpecificationII*

SpecificationII can be seen as a composition of channels that are specified using *SpecificationII*^{single}. To create a state of *SpecificationII* from states of multiple instances of *SpecificationII*^{single} for different channels, the channel variables of these channels can directly be copied to the corresponding channels of *SpecificationII* because there are no conflicts because each set of channel variables occurs only in a single instance of *SpecificationII*^{single}. However, the user variables and global variables need to be merged (see Fig. 4.27 on page 135). For example, the value of the variable *UserPreimageInventory[u]* for user u in *SpecificationII* can be defined as the union of all values of the same variable in instances of *SpecificationII*^{single}. This means that, in *SpecificationII*, a user u knows all preimages that the user u knows in a specific channel. If values of variables are changed during a mapping from *SpecificationII*^{single} to *SpecificationII*, it needs to be ensured that all behaviors of a channel in *SpecificationII*^{single} are also possible in *SpecificationII*. To show this, Lemma 10 shows that every step of the modules *AbstractEnforcement* and *HTLCUser* in *SpecificationII*^{single} is also possible if the user variables and global variables contain additional values from other channels.

Lemma 10. *A step of AbstractEnforcement or HTLCUser in SpecificationII*^{single} *is also possible if the variables that are reduced by* m_c *and are used in SpecificationII*^{single} *(i.e., UserPreimageInventory[u], UserLatePreimages[u], UserNewPayments[u], UserPayments[u], UserExtBalance[u], UserChannelBalances[u], Messages) contain additional values that are filtered out by* m_c *(see Definition 19).*

We prove Lemma 10 in the appendix in Section A.3.12 and sketch the proof idea here. In the proof, we discuss how those variables that are changed by the mapping m_c are used in each action of the modules *AbstractEnforcement* and *HTLCUser*. We prove that a step $s \rightarrow t$ of the modules *AbstractEnforcement* and *HTLCUser* is also possible if those variables contain additional values that are removed by the mapping m_c. To structure the proof, we separately discuss unprimed usages of values that refer the value of a variable in state s and primed usages of values that refer to a variable's value in state t. To illustrate the proof method, we give two examples.

Example 4.15. Consider the condition: $preimage \in UserPreimageInventory[u]$. If this condition is true in step $s \rightarrow t$, then it must also be true if the set *UserPreimageInventory[u]* contains additional values.

Example 4.16. Consider the following condition that specifies that the variable *UserPayments[u]* is updated so that the state of all payments in *processedPayments* is advanced to `processed`.

$$
\begin{array}{l}
\textsc{let}\ fulfilledHTLCs \triangleq \{htlc \in Vars.OutgoingHTLCs : htlc.hash = message.data.preimage\} \\
\quad processedPayments \triangleq \{payment \in Payments : \\
\qquad\qquad \land\ payment.state = \text{“NEW”} \\
\qquad\qquad \land \exists\, htlc \in fulfilledHTLCs : htlc.id = payment.id \\
\qquad \} \\
\textsc{in}\ \ Payments' = (Payments \setminus processedPayments) \cup \\
\qquad\qquad \{[payment\ \textsc{except}\ !.state = \text{“PROCESSED”}] \\
\qquad\qquad\quad : payment \in processedPayments \\
\qquad\qquad \}
\end{array}
$$

This is a (slightly simplified) excerpt of the definition of the action *ReceiveHTLCPreimage* of the module *HTLCUser*. If this condition about the values of *Payments* and *Payments*′ is true in step $s \rightarrow t$, then it must also be true if the sets *Payments* and *Payments*′ contain the same additional payments and the set *processedPayments* does not include those additional payments. The set *processedPayments* contains only payments for which an HTLC exists in *Vars.OutgoingHTLCs*. Such payments for which an HTLC exists in *Vars.OutgoingHTLCs* or *Vars.IncomingHTLCs* are retained in the set *Payments* by the mapping m_c. Thus, any payments that are removed by the mapping m_c will not be in the set *processedPayments*.

□

By proving Lemma 10, we have shown that each payment channel c specified by *SpecificationII*^{single} can be combined with other payment channels in *SpecificationII* so that each behavior of the payment channel c in *SpecificationII*^{single} is still possible after combining the channel with other payment channels. Our next step is to combine Lemmas 8 to 10 to show that *SpecificationI*^{time} is a refinement of *SpecificationII*.

4.3.3.4. Refinement Mapping from *SpecificationI*^{time} to *SpecificationII*

In this subsection, we define the mapping g. Note that, technically, we do not define the mapping g to compose a state of *SpecificationII* from multiple instances of *SpecificationII*^{single}. In Fig. 4.27, this would correspond to mapping each channel from *SpecificationI*^{time} to *SpecificationI*$^{time,\ single}$ to *SpecificationII*^{single} to *SpecificationII*. Instead, the mapping g is defined to directly map each state of *SpecificationI*^{time} to a state of *SpecificationII*. This is equivalent because the global variables and user variables have the same values in a state of *SpecificationI*^{time} and the corresponding state of *SpecificationII*. Only the values of the channel variables need to be changed because the enforcement layer of payment channels is abstracted.

Definition 21 (Refinement mapping g from *SpecificationI*^{time} to *SpecificationII*)**.** Define the mapping g from the state space of *SpecificationI*^{time} to the state space of *SpecificationII*, as follows. For a state s of *SpecificationI*^{time}, let $g(s)$ be a state of *SpecificationII* with the following value to variable assignments:

- Each variable v of the global variables and the user variables is assigned the variable's value in state s because these variables are left unchanged by the mapping f.
- Each variable v of the channel variables and the ⟨channel and user⟩ variables for channel c is assigned the value of the variable after mapping the state s by the mappings m_c and f, i.e., the value of $f(m_c(s)).v$.

Remark. The definition of the value of $g(s).v \coloneqq s.v$ for a state s of *SpecificationI*^{time} and a global variable v or a user variable v is direct and simple. An equivalent definition could be given by defining the value of $g(s).v$ based on the values of $f(m_c(s))$, i.e. based on a state of *SpecificationII*^{single}. For example, the value of the variable *Messages* in a state $g(s)$ could also be obtained by merging the sets of messages in the instances of *SpecificationII*^{single} of all channels. Analogously, the value of the variable *LedgerTime* in a state $g(s)$ could be obtained by choosing the maximal value of all values of the variable *LedgerTime* in instances of *SpecificationII*^{single} of all channels.

Finally, Theorem 3 states that the mapping g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII* which implies that *SpecificationI*^{time} is a refinement of *SpecificationII*.

Theorem 3. *The mapping g is a refinement mapping from SpecificationI^{time} to SpecificationII.*

We prove Theorem 3 in the appendix in Section A.3.13 and provide an informal version of the proof here. In general, to prove that a mapping is a refinement mapping from system S_1 to system S_2 the following properties need to be shown [AL91]:

- The external variables of every behavior of system S_1 are unchanged by the refinement mapping. (P1)
- A behavior σ_2, to which a behavior σ_1 of system S_1 is mapped, is a valid behavior of system S_2. This is shown by showing the following properties:
  - The initial state of σ_1 is mapped to an initial state of system S_2. (P2)
  - Each step in σ_1 is mapped to a step that is valid in system S_2. (P3)
  - The behavior σ_2 fulfills the liveness condition of system S_2. (P4)

We use this proof scheme to show that the mapping g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII*. We refer to Fig. 4.27 on page 135 that visualizes the mapping g and the related mappings as well as the groups of variables used in the two specifications. We explain how we prove that each of the four properties hold. By [AL91], Theorem 3 follows from these properties.

Property P1 holds because the external variables (the variables that occur in the security property) are user variables which are, by definition of the mapping g, unchanged by the mapping g.

Property P2 is fulfilled for the following reasons. The mapping g maps the values of global variables and user variables to the same values. The values of channel variables are changed by the mapping f. By definition of the mapping f, the mapping does not change the values of initial states. Therefore, the mapped values of all variables that occur in *SpecificationII* are equal to the values of the corresponding variables in *SpecificationI*^{time}. Because the $Init$ predicates of both specifications are equal (except that there are fewer variables in *SpecificationII*), it holds that property P2 is fulfilled.

To show that property P3 holds, we have to show that the mapping g maps each step in *SpecificationI*^{time} to a step of *SpecificationII*. We show this by distinguishing steps of module *LedgerTime* and steps of the modules *PaymentChannelUser* and *HTLCUser*. To show that a step $s \rightarrow t$ of module *LedgerTime* in *SpecificationI*^{time} is mapped to a step $g(s) \rightarrow g(t)$ of module *LedgerTime* in *SpecificationII*, we show that the step $g(s) \rightarrow g(t)$ fulfills the definition of the *Next* action of module *LedgerTime* in *SpecificationII*. By the definition of module *LedgerTime*, in the step $g(s) \rightarrow g(t)$ it needs to hold that ...

- ... all variables other than the variable *LedgerTime* are unchanged.

  By definition of the module *LedgerTime* in *SpecificationI*^{time}, the values of all variables other than *LedgerTime* and *TxAge* are unchanged. The variable *TxAge* does not occur in *SpecificationII* because the specification abstracts from transactions on the blockchain. By definition of the mapping g, the mapped values of global variables and user variables are unchanged. Because the definition of the mapping f does not depend on the value of the variable *LedgerTime*, all channel variables in state $g(t)$ are mapped to the same values as in state $g(s)$. Therefore, all variables other than *LedgerTime* are unchanged in the step $g(s) \rightarrow g(t)$.

- ... the value of the variable *LedgerTime* increases to any larger value.

  Because the mapping g does not change the value of the variable *LedgerTime* and, by definition of module *LedgerTime*, the value of *LedgerTime* increases in the step $s \rightarrow t$, the value of *LedgerTime* increases also in $g(s) \rightarrow g(t)$. Here, it is relevant that *SpecificationII* is, in contrast to *SpecificationI*^{time}, not a time-optimized specification. This proof step would be more complicated if *SpecificationII* was a time-optimized specification (see Section 4.1.3) because then the new value of *LedgerTime* in *SpecificationII* would have to be included in the set $relZTP$ and may not simply be any larger value than the previous value.

- ... the value of the variable *LedgerTime* does not skip a time bound, i.e. it holds for all time bounds b that if $b \geq g(s).LedgerTime$ then $b \geq g(t).LedgerTime$.

  By definition of module *LedgerTime*, the step $s \rightarrow t$ does not skip any time bounds of *SpecificationI*^{time}. The time bounds in specifications *SpecificationI*^{time} and *SpecificationII* are defined as the union of time bounds specified by each channel in the specifications. Thus, it suffices to prove that it holds for each channel that each time bound

that exists in *SpecificationII* also exists in *SpecificationI*^{time} and is therefore not skipped in the step $s \rightarrow t$. We prove this by showing that the time bounds for each channel c in a step $s \rightarrow t$ in *SpecificationI*^{time} are the same for a step $m_c(s) \rightarrow m_c(t)$ in *SpecificationI*$^{time,\ single}$. Because the refinement mapping f maps each step $m_c(s) \rightarrow m_c(t)$ of module *LedgerTime* to a valid step $f(m_c(s)) \rightarrow f(m_c(t))$ of *SpecificationII*^{single} (by Lemma 9), there cannot exist time bounds in *SpecificationII*^{single} that do not exist in *SpecificationI*$^{time,\ single}$. It follows that there are no time bounds in *SpecificationII* that do not exist in *SpecificationI*^{time} and, thus, each step of module *LedgerTime* in *SpecificationI*^{time} can be mapped to a step of module *LedgerTime* in *SpecificationII*.

Next, we show that a step $s \rightarrow t$ of the modules *PaymentChannelUser* or *HTLCUser* in *SpecificationI*^{time} is mapped to a step $g(s) \rightarrow g(t)$ of the modules *AbstractEnforcement* or *HTLCUser* in *SpecificationII*. Let channel c be the channel and user u be the user which is updated in step $s \rightarrow t$. We show that all variables of *SpecificationII* are updated in the step $g(s) \rightarrow g(t)$ according to the definitions of the modules *AbstractEnforcement* or *HTLCUser*. We distinguish global variables, user variables for user u, channel variables for channel c, user variables for users $u' \neq u$, and channel variables for channels $c' \neq c$:

- The channel variables for channels $c' \neq c$ and user variables for users $u' \neq u$ are unchanged in the step $s \rightarrow t$ and, thus, these variables are unchanged in the step $g(s) \rightarrow g(t)$ as required by the definitions of the modules *AbstractEnforcement* or *HTLCUser*.
- Considering the channel variables and ⟨channel and user⟩ variables for channel c: By Lemma 8, the step $m_c(s) \rightarrow m_c(t)$ is a step of the modules *PaymentChannelUser* or *HTLCUser* in *SpecificationI*$^{time,\ single}$. The refinement mapping f maps this step to a step $f(m_c(s)) \rightarrow f(m_c(t))$ of the modules *AbstractEnforcement* or *HTLCUser* in *SpecificationII*^{single} (see Lemma 9). By definition of the mapping g, the channel variables for channel c have the same values in the step $g(s) \rightarrow g(t)$ as in the step $f(m_c(s)) \rightarrow f(m_c(t))$. Because the same modules *AbstractEnforcement* or *HTLCUser* are used in *SpecificationII*^{single} and *SpecificationII*, the values of the channel variables for channel c are updated in step $g(s) \rightarrow g(t)$ as required by the modules *AbstractEnforcement* and *HTLCUser*.
- The same holds for user variables for user u except that these variables might contain in the step $g(s) \rightarrow g(t)$ in *SpecificationII* additional values compared to the step $f(m_c(s)) \rightarrow f(m_c(t))$ in *SpecificationII*^{single}. Depending on the specific variable, these additional values might contain preimages or payments that are not related to channel c but to other channels and are, therefore, removed by the mapping m_c. By Lemma 10, the user variables for user u are updated in the step $g(s) \rightarrow g(t)$ according to the modules *AbstractEnforcement* or *HTLCUser* in *SpecificationII* although they might contain additional values compared to their values in the step $f(m_c(s)) \rightarrow f(m_c(t))$.
- There are two global variables in *SpecificationII*: *Messages* and *LedgerTime*. For the variable *Messages* the same holds as for user variables for user u and by Lemma 10 the step $g(s) \rightarrow g(t)$ updates the variable *Messages* as allowed by the modules

AbstractEnforcement or *HTLCUser*. While the value of the variable *LedgerTime* does not change in the step $s \rightarrow t$ and, by definition of the mapping g, not in the step $g(s) \rightarrow g(t)$, the value of *LedgerTime* might be lower in the step $f(m_c(s)) \rightarrow f(m_c(t))$ as in the step $g(s) \rightarrow g(t)$ because the mapping m_c might reduce its value. While we have shown that the step $f(m_c(s)) \rightarrow f(m_c(t))$ is a step of the modules *AbstractEnforcement* or *HTLCUser*, it is left to show that the step $g(s) \rightarrow g(t)$ is a step of the modules *AbstractEnforcement* or *HTLCUser* although the value of the variable *LedgerTime* might be greater than in the step $f(m_c(s)) \rightarrow f(m_c(t))$. As shown in the proof in Section A.3.13, it holds that for all states s and t in which the value of *LedgerTime* is reduced by the mapping m_c there exist equivalent states s' and t' in which the value of *LedgerTime* is unchanged by the mapping m_c. It follows that $f(m_c(s')) \rightarrow f(m_c(t'))$ is a step of the modules *AbstractEnforcement* or *HTLCUser* in *SpecificationII*^{single} in which all variables except *LedgerTime* have the same values as in the step $f(m_c(s)) \rightarrow f(m_c(t))$. Because the value of *LedgerTime* is the same in step $g(s) \rightarrow g(t)$ as in the step $f(m_c(s')) \rightarrow f(m_c(t'))$, the step $g(s) \rightarrow g(t)$ is a valid step of the modules *AbstractEnforcement* or *HTLCUser*.

Property P4 holds because *SpecificationII* contains the same liveness condition as *SpecificationI*^{time} that only behaviors are valid that end in a state in which all users have withdrawn their balance.

From the properties P1 to P4 it follows that g is a refinement mapping from *SpecificationI*^{time} to *SpecificationII*. □

Having proven Theorem 3, we have shown that *SpecificationI*^{time} implements *SpecificationII*. Combined with Theorem 2 that states that *SpecificationI* implements *SpecificationI*^{time}, we conclude that *SpecificationI* implements *SpecificationII*.

4.4. Abstraction of Payment Channels

We introduce an additional abstraction of the specification of Lightning to further reduce the specification's state space. The idea behind the abstraction from *SpecificationII* to *SpecificationIII* is to abstract a payment channel as a whole. In the following, we explain the idea behind *SpecificationIII* by looking at what a payment channel does from a user's perspective and then we give an overview of the remainder of this section. To discuss what a payment channel does on an abstract level, we distinguish the perspectives of the sender of a payment, the receiver of a payment, and a user who forwards a payment:

From the perspective of the sender of a payment, the sender adds the payment to a payment channel and the payment channel will resolve the payment either by processing the payment or by aborting the payment. Even if the payment is a multi-hop payment, it is guaranteed that the payment is only processed if the receiver of the payment has processed the payment; otherwise, the payment will be aborted. If the payment is processed, the sender's balance is reduced by the payment's amount.

From the perspective of the receiver of a payment, a payment appears in one of the receiver's payment channels. The receiver can decide how to resolve the payment: Either the receiver processes the payment and the payment's amount is added to the receiver's balance or the receiver aborts the payment.

A user who forwards a payment can be seen as an intermediate receiver and an intermediate sender of the payment. From the perspective of a user who forwards a payment, a payment appears in one of the user's payment channels. The payment includes information specifying the channel through which it is to be forwarded. The user can either forward the payment by adding the payment to the specified channel or abort the payment. If the user forwards the payment, the payment in the channel on which it has been forwarded will either be processed or aborted. If the payment is processed, the forwarding user processes the payment in the channel in which the user has received the payment. If the payment is aborted, the forwarding user aborts the payment in the channel in which the payment has been received. If a payment is processed in a payment channel, the balance of the forwarding user inside the channel is increased or decreased by the payment's amount depending on whether the payment is incoming or outgoing in the payment channel. If the forwarding user follows the described behavior, the balance that the forwarding user has in total in both payment channels remains unchanged whether the payment is processed in both channels or aborted in both channels.

To summarize, a payment channel can be seen as a component that models how payments and user balances change. A user can add a payment to a payment channel and the payment channel forwards the payment to the other user of the payment channel. The receiving user can decide how to resolve the payment. The payment is resolved by either processing the payment and updating the balances of the users in the channel or by aborting the payment. A payment can only be processed by the ultimate receiver of the payment or by a forwarding user who has added the payment to a payment channel and the payment was processed in this payment channel.

The goal of a protocol for a payment channel network is to synchronize the state of a payment in the payment channels along a payment's route: Finally, either a payment should be processed in all payment channels or aborted in all payment channels. To achieve this synchronization of a payment's state, Lightning uses mainly three mechanisms: Preimages, failure messages, and a global clock.

In Lightning, each payment is associated with a corresponding preimage. The preimage is used to ensure that a payment can only be processed by the ultimate receiver of the payment or by a forwarding user who has added the payment to a payment channel and the forwarded payment was processed: Initially, only the receiver of a payment knows the preimage associated with the payment. The receiver can only process the payment in a payment channel by sharing the preimage with the other user in the payment channel. Thereby, the other user can again share the preimage to process the payment in another channel. The preimage is forwarded between the users along a payment's route until it reaches the sender of a payment and the payment is processed in all channels on the payment's route. If all users follow the protocol and forward the preimage, this mechanism achieves that, once the receiver of a payment decides to process the payment,

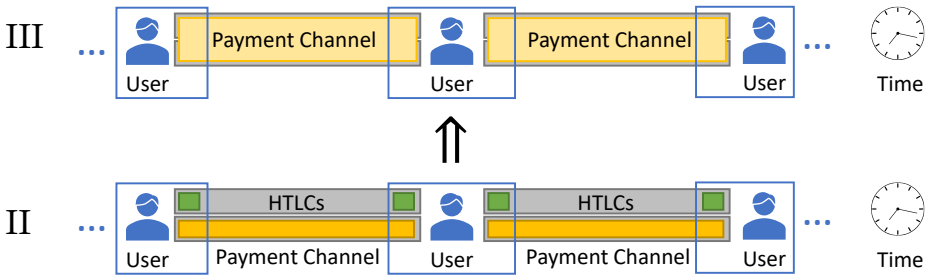

Figure 4.32.: System overview of *SpecificationII* and *SpecificationIII*. In *SpecificationII*, each payment channel is implemented by an instance of the module *AbstractEnforcement* (yellow box) and one instance of the module *HTLCUser* per user in the channel. In *SpecificationIII*, each payment channel is implemented by an instance of the module *ChannelAbstraction* (light yellow box).

all channels along the payment's route are synchronized to the outcome that the payment is processed.

When a user receives an incoming payment, the user can decide to abort the payment instead of forwarding the payment or, if the user is the payment's receiver, instead of processing the payment by sharing the preimage. A payment is aborted in Lightning by sending a message that the payment has failed. This message is forwarded along the payment's route to the payment's sender thereby aborting the payment in all payment channels and thereby synchronizing the payment's state in all payment channels.

If neither preimages are shared between users nor messages that the payment has failed, a payment is aborted in all payment channels depending on time. A payment has in each payment channel an absolute timelock that depends on the value of a global clock. The timelocks are decreasing from the sender's payment channel to the payment channel of the payment's receiver. If the timelock of a payment in a payment channel has passed, the payment is resolved in that payment channel as aborted. After the timelock of the payment in the sender's payment channel has passed, the payment is aborted in all payment channels. Thereby, a global clock is used to synchronize a payment's state in all payment channels.

In this section, we model a payment channel based on this abstract view with the module *ChannelAbstraction* (see Fig. 4.32). The module *ChannelAbstraction* implements a payment channel with a set of forwarded payments over the payment channel, the balances of the two users in the channel, and a state that indicates whether the payment channel is open or closed. The module contains actions to forward payments and to resolve payments. As described above, how a payment can be resolved is determined by the preimages that a user has, by the messages for failed payments that a user has received, and by the timelock of a payment. We further describe the module *ChannelAbstraction* in Section 4.4.2.

Remark. The synchronization of how a payment is resolved works well only if all users along a payment's route are honest. If one or multiple users are dishonest, the payment might be resolved in some channels as processed and in other channels as aborted. For example, a dishonest forwarding user might not resolve an incoming payment as processed

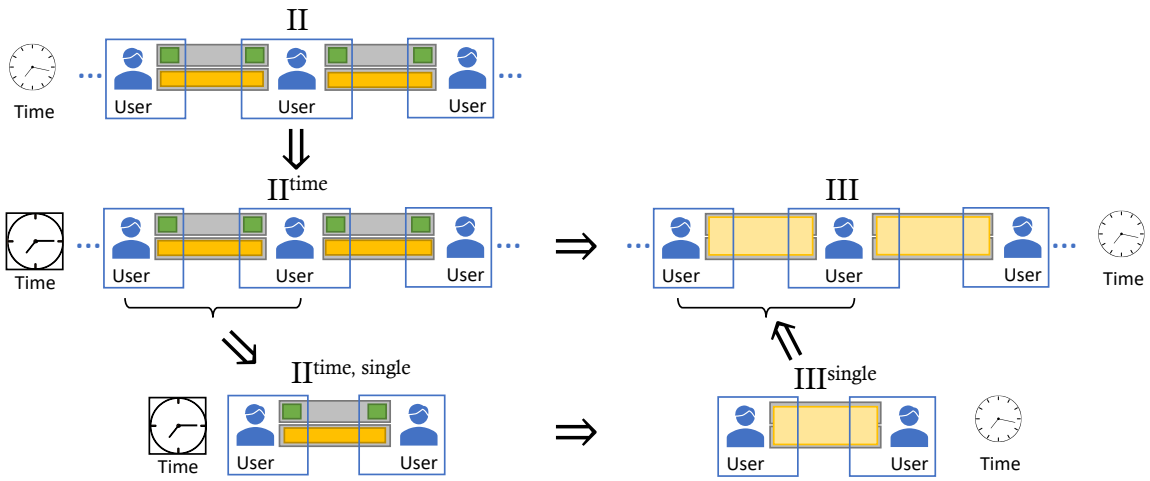

Figure 4.33.: Approach to check the abstraction from *SpecificationII* to *SpecificationIII*. We use the time abstraction (see Section 4.1) to show that *SpecificationII* implements $SpecificationII^{time}$ and, then we decompose $SpecificationII^{time}$ and show that each single channel specification $SpecificationII^{time,\ single}$ implements the single channel specification $SpecificationIII^{single}$. The composition of payment channels specified by $SpecificationIII^{single}$ is a payment channel network as specified by *SpecificationIII*.

before the payment's timelock even though the user knows the payment's preimage. In such a situation, the state of a payment is not synchronized in all channels along the payment's route. However, as we show in this chapter, Lightning ensures that an honest user does not lose funds even if a payment is resolved in two conflicting ways in the user's payment channels.

In *SpecificationII*, each payment channel is implemented using an instance of the module *AbstractEnforcement* and, per user, one instance of the module *HTLCUser*. We show in this section, that the implementation of a payment channel used by *SpecificationII* implements the module *ChannelAbstraction* that is used in *SpecificationIII*. The approach to show this implementation is depicted in Fig. 4.33. To facilitate model checking, we first define a time-optimized version of *SpecificationII* that we refer to as $SpecificationII^{time}$. We show in Section 4.4.1 that *SpecificationII* implements $SpecificationII^{time}$ by applying Theorem 1 that was defined in Section 4.1 on page 112. We present *SpecificationIII* and the module *ChannelAbstraction* in more detail in Section 4.4.2. Then, we use in Section 4.4.3 the decomposition approach to show that $SpecificationII^{time}$ implements *SpecificationIII*. To this end, we decompose $SpecificationII^{time}$ and *SpecificationIII* into specifications for single payment channels that are called $SpecificationII^{time,\ single}$ and $SpecificationIII^{single}$. We specify a refinement mapping from $SpecificationII^{time,\ single}$ to $SpecificationIII^{single}$ and show by model checking the correctness of this refinement mapping that $SpecificationII^{time,\ single}$ implements $SpecificationIII^{single}$. We prove that payment channels specified by $SpecificationIII^{single}$ can be combined with other channels in *SpecificationIII*. Finally, we use these results to prove that $SpecificationII^{time}$ implements *SpecificationIII*. Because the proof is analogous to the proof that *SpecificationI* implements *SpecificationII*, we do not repeat a description of the proof ideas and, instead, present only the main statements.

4.4.1. Application of Theorem 1 for *SpecificationII*

SpecificationII is defined as a real-time specification in which time can advance by arbitrary natural numbers instead of using the improved model of time. This facilitates the refinement mapping g from *SpecificationI*^{time} to *SpecificationII* because it ensures that every step that advances time in *SpecificationI*^{time} is also allowed in *SpecificationII*. To allow for an efficient model checking, we apply the time-optimization introduced in Section 4.1 by defining *SpecificationII*^{time}. We prove that Theorem 1 applies also to the real-time specification *SpecificationII* and the corresponding time-optimized specification *SpecificationII*^{time}. It follows that *SpecificationII* implements *SpecificationII*^{time} and, by transitivity, *SpecificationI* implements *SpecificationII*^{time}.

To prove that Theorem 1 can be applied to *SpecificationII*, we prove that the assumptions for Theorem 1 hold for *SpecificationII* (analogously to Section 4.3.1). Because *SpecificationII* has a lower complexity (e.g., only *LedgerTime* as a single clock), the proofs for *SpecificationII* have a lower complexity than those for *SpecificationI*. The proof ideas are the same as for the corresponding proofs in Section 4.3.1. Thus, we do not describe the proof ideas again. The full proofs can be found in the appendix (see Sections A.3.14 to A.3.16).

We start by proving that the time bounds specified in module *SpecificationII* meet Assumption 4.1:

Lemma 11. *In SpecificationII it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T_d^y(s))$$

We prove Lemma 11 in the appendix in Section A.3.14. The proof is analogous to the proof of Lemma 4 that is sketched in Section 4.3.1.3. □

Lemma 12. $relZTP^x$ *of SpecificationII*^{time} *meets Assumption 4.2, i.e.* $\forall x \in \mathcal{X}, s \in \hat{\Sigma} : ZTP^x(s) \subseteq relZTP^x(s)$.

We prove Lemma 12 in the appendix in Section A.3.15. The proof is analogous to the proof of Lemma 5 that is sketched in Section 4.3.1.3. □

The next assumption to show is that *SpecificationII* fulfills Assumption 4.3, i.e. that for a $NextII$ step from state s to state t it holds for all clocks $x \in \mathcal{X}$ that $relZTP^x(t) \subseteq relZTP^x(s)$.

Lemma 13. *For any two states* s, t *of SpecificationII and all clocks* $x \in \mathcal{X}$ *(i.e., the clock LedgerTime), it holds that*

$$\textit{NextII} \Rightarrow relZTP^{x\prime} \subseteq relZTP^x$$

We prove Lemma 13 in the appendix in Section A.3.16. The proof is analogous to the proof of Lemma 6 that is sketched in Section 4.3.1.3. □

Finally, we show that *SpecificationII* implements $SpecificationII^{time}$.

Theorem 4.

$$SpecificationII \Rightarrow SpecificationII^{time}$$

Proof: From Lemmas 11 to 13 it follows that *SpecificationII* fulfills the assumptions Assumptions 4.1 to 4.3. Therefore, we can apply Theorem 1 to *SpecificationII*. Because $SpecificationII^{time}$ is defined according to the definition of the time-optimized specification $Spec_{\hat{S}}$ in Section 4.1.3, this implies that *SpecificationII* implements $SpecificationII^{time}$. □

4.4.2. *SpecificationIII* and Module *ChannelAbstraction*

While each payment channel was modeled in *SpecificationII* by an instance of the module *AbstractEnforcement* and two instances of the module *HTLCUser* (one per user), we abstract these three instances in *SpecificationIII* by a single instance of the module *ChannelAbstraction*. The idea of the module *ChannelAbstraction* is to provide a simple implementation of the interface between the two users of payment channel and the payment channel itself. To this end, the module *ChannelAbstraction* abstracts the states of HTLCs and represents the state of a payment only with the states NEW, PROCESSED, and ABORTED. Whether a payment can be processed or aborted is determined based on the information available to each channel. For example, a payment can only be processed if the payment's receiver has the corresponding preimage. A payment can be aborted if the payment has timed out or if the receiver sends a message to fail the payment.

The lifecycle of a payment channel in *SpecificationIII* is as follows: A payment channel can be opened using the action *OpenChannel* of the module *ChannelAbstraction*. For each payment, the payment's sender requests an invoice from the payment's receiver which is specified in the module *ChannelAbstraction* by actions analogous to the respective actions of the module *HTLCUser*. After a payment has been added to a payment channel, a record for the payment with state NEW is stored in the variable *ChannelForwardedPayments[c]* for channel c. The action *ResolvePayments* of the module *ChannelAbstraction* describes under which circumstances a payment can be resolved and how an update to a payment's state affects other variables. If a payment is processed, the preimage for the payment is added to the variable *UserPreimageInventory[u]* of the user u who has sent or forwarded the payment and the balances of the users are updated. The definition of the action *ResolvePayments* is complex because the action needs to cover dishonest behavior of users which includes, for example, payments being fulfilled after they were timed out or payments being timed out although the user to whom the payment was forwarded knows the payment's preimage. Closing of a payment channel is also modeled by the action *ResolvePayments* because the state of payments can change in the step that closes a payment channel.

To give an impression of the specification of the module *ChannelAbstraction*, we give examples for the conditions used by the action *ResolvePayments* of the module *ChannelAbstraction* to determine how a payment can be resolved: A payment can only be processed in a channel if the user to whom the payment was forwarded to knows the preimage associated with the payment. A payment cannot be processed and must be aborted if

- both users of the channel are honest and the value of the global clock (variable *LedgerTime*) is greater than or equal to the payment's absolute timelock plus the forward delta (the difference between the timelocks of the payment in adjacent payment channels), or
- a message to fail the payment was sent over the channel from the user to whom the payment was forwarded to, or
- the preimage associated with the payment was shared over the channel but the payment has not yet been resolved.

The last condition can only be fulfilled if at least one of the users in the channel is dishonest. A payment can only be aborted in a payment channel if

- the value of the global clock is greater than or equal to the payment's absolute timelock, or
- the payment channel is the last hop on the payment's route, or
- the user to whom the payment was forwarded to has received a message to fail the payment.

The definition of the action *ResolvePayments* has more conditions and is complex, in particular, because how a payment is resolved can be affected by dishonest behavior.

4.4.3. *SpecificationII*^{time} Implements *SpecificationIII*

To show that *SpecificationII*^{time} implements *SpecificationIII*, we use the decomposition approach (see Section 4.2) that we have already used in Section 4.3.3. Because the proof is analogous to the proof in Section 4.3.3, we present in the following only the lemmas and basic ideas but do not repeat further details.

4.4.3.1. Channels in *SpecificationII*^{time} Implement *SpecificationII*$^{time,\ single}$

We first decompose *SpecificationII*^{time} into single channels of which each is specified by specification *SpecificationII*$^{time,\ single}$. *SpecificationII*$^{time,\ single}$ specifies the behavior of a single payment channel and includes the module *ChannelEnvironment*, which is an environment module that abstracts the effects that other payment channels can have on the modeled payment channel. The mapping m_c^{II} specifies a mapping that maps a channel c in specification *SpecificationII*^{time} to a state of *SpecificationII*$^{time,\ single}$. The following lemma states that with the mapping m_c^{II}, each behavior of *SpecificationII*^{time} is

mapped to a valid behavior of *SpecificationII*$^{time,\ single}$. This means that the single channel specification *SpecificationII*$^{time,\ single}$ describes all behaviors of a payment channel as if the payment channel were combined with other channels.

Lemma 14. *For each channel $c \in C$, the mapping m_c^{II} maps each behavior $\sigma = \langle \sigma_0, \sigma_1, \sigma_2, ... \rangle$ of SpecificationII^{time} to a behavior $\langle m_c(\sigma_0), m_c(\sigma_1), m_c(\sigma_2), ... \rangle$ of SpecificationII$^{time,\ single}$ of channel c.*

We prove Lemma 14 in the appendix in Section A.3.17. The proof is analogous to the proof of Lemma 8 that is sketched in Section 4.3.3.1. □

4.4.3.2. *SpecificationII*$^{time,\ single}$ Implements *SpecificationIII*^{single}

We specify a refinement mapping f^{II} from *SpecificationII*$^{time,\ single}$ to *SpecificationIII*^{single}. This refinement mapping calculates the state of each payment that is forwarded over the payment channel (stored in *ChannelForwardedPayments[c]* in *SpecificationIII*^{single}) based on the state of the corresponding HTLC in the variables of the users in *SpecificationII*$^{time,\ single}$. The refinement mapping f^{II} is formally defined in TLA$^+$in *SpecificationII* in [GH26b]. We check that the refinement mapping f^{II} is actually a refinement mapping by checking that the application of the mapping f^{II} on states of any behavior of *SpecificationII*$^{time,\ single}$ results in a valid behavior of *SpecificationIII*^{single}. We state this in the following lemma and verify the lemma using model checking.

Lemma 15. *SpecificationII*$^{time,\ single}$ *for channel $c \in C$ is a refinement of SpecificationIII*^{single} *with the refinement mapping f^{II}.*

We check the correctness of this lemma using model checking (see Section 4.7). We model check *SpecificationII*$^{time,\ single}$ and verify that applying the mapping f^{II} to each step $s \rightarrow t$ of *SpecificationII*$^{time,\ single}$ leads to steps $f^{II}(s) \rightarrow f^{II}(t)$ of *SpecificationIII*^{single} and that each behavior of *SpecificationII*$^{time,\ single}$ is mapped to a behavior of *SpecificationIII*^{single}.

4.4.3.3. Composition of *SpecificationIII*^{single} Implements *SpecificationIII*

We show that channels modeled by the single channel specification *SpecificationIII*^{single} can be composed to a payment channel network as defined by *SpecificationIII*. To this end, we prove the following lemma in Section A.3.18.

Lemma 16. *A step of ChannelAbstraction in SpecificationIII*^{single} *is also possible if the variables that are reduced by m_c^{II} and are used in SpecificationIII contain additional values that are filtered out by m_c^{II}.*

We prove Lemma 16 in the appendix in Section A.3.18. The proof is analogous to the proof of Lemma 10 that is sketched in Section 4.3.3.3. □

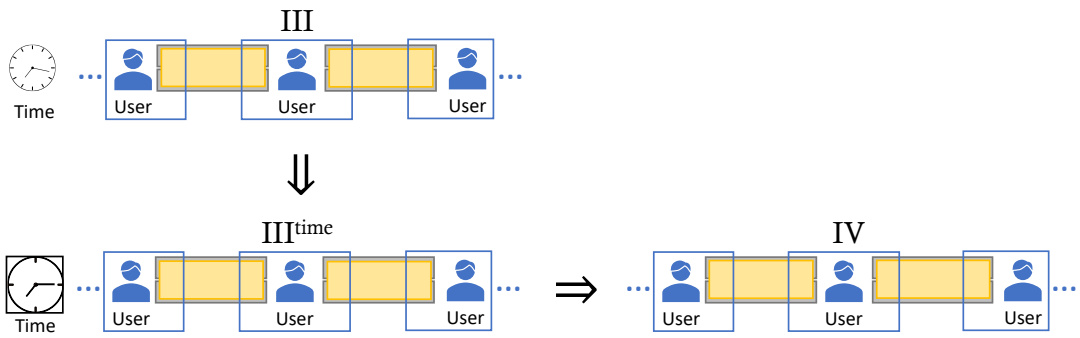

Figure 4.34.: Approach to check the abstraction from *SpecificationIII* to *SpecificationIV*. We use the time abstraction (see Section 4.1) to show that *SpecificationIII* implements $SpecificationIII^{time}$ and verify by model checking that $SpecificationIII^{time}$ implements *SpecificationIV*.

4.4.3.4. Refinement Mapping from $SpecificationII^{time}$ to *SpecificationIII*

Finally, we show with Theorem 5 that $SpecificationII^{time}$ implements *SpecificationIII* with the refinement mapping g^{II}. We prove this theorem in the appendix in Section A.3.19 based on Lemmas 14 to 16. The proof is analogous to the proof for the refinement mapping from $SpecificationI^{time}$ to *SpecificationII*.

Theorem 5. *The function g^{II} is a refinement mapping from $SpecificationII^{time}$ to SpecificationIII.*

We prove Theorem 5 in the appendix in Section A.3.19. The proof is analogous to the proof of Theorem 3 that is sketched in Section 4.3.3.4. □

4.5. Abstraction of Inter-Channel Communication

While *SpecificationIII* abstracted payment channels, the specification of these payment channels uses only information that is local to each payment channel. This resembles the conditions that apply to the actual Lightning protocol because, in the real world, each user can access only the user's own data. For example, whether a payment can be processed is determined by checking whether the user to whom the payment was forwarded to has the corresponding preimage. Besides preimages, *SpecificationIII* contains also sets of failed payments and models time to determine which payments can be aborted because they have failed to be forwarded or have timed out. In principle, these mechanisms are not required to model a payment channel network in which channels have global knowledge because the decision how a payment may be resolved can be made based on how the payment has been resolved in other payment channels.

We use this idea to further abstract the specification of Lightning in *SpecificationIV* which models a payment channel network without explicit communication between channels. Instead, the possible states of each payment in a payment channel are determined based on

the payment's state in other channels. This is a fundamental difference from the previous abstractions of Lightning because in *SpecificationIV* the behavior of a payment channel is specified depending on the variables of the complete payment channel network while in *SpecificationIII* each payment channel was specified depending only on its own variables and the variables of the users of the payment channel.

To give examples for the conditions for how payments are resolved in *SpecificationIV*, the following informal conditions are some examples assuming that users are honest:

- A payment can be processed in channel c if channel c is the final hop of the payment or if the payment is processed in the next channel on the route to the payment's receiver.
- A payment may be aborted in channel c if channel c is the final hop.
- If a payment was aborted in a channel that is closer to the sender of the payment, the payment must be aborted.

The conditions specified in *SpecificationIV* are more complex because they also handle the cases for dishonest users. If users are dishonest, it can, for example, happen that the state of a forwarded payment in channel c changes from PROCESSED to ABORTED. *SpecificationIV* ensures that users who forward a payment do not lose funds by ensuring that an honest user who has received a payment in one channel and forwards the payment in another channel does not end in a state in which the incoming payment is aborted and the outgoing payment is processed.

SpecificationIV models payment channels using a subset of the actions of the module *ChannelAbstraction*. The main difference from *SpecificationIII* is that the possible future states of a payment are determined in *SpecificationIII* by using local knowledge of preimages, failed payments, and time whereas *SpecificationIV* uses global knowledge of the states of a payment in other channels.

To check that *SpecificationIII* implements *SpecificationIV*, we first create the time-optimized specification $SpecificationIII^{time}$ for *SpecificationIII* in Section 4.5.1. In Section 4.5.2, we define a refinement mapping from $SpecificationIII^{time}$ to *SpecificationIV* and model check that $SpecificationIII^{time}$ implements *SpecificationIV*.

4.5.1. Application of Theorem 1 for *SpecificationIII*

SpecificationIII is defined as a real-time specification in which time can advance by arbitrary natural numbers instead of using the optimized model of time. As in the refinement steps before, we specify a time-optimized specification $SpecificationIII^{time}$ and prove that *SpecificationIII* implements $SpecificationIII^{time}$ by applying Theorem 1 that was defined on page 112.

To prove that Theorem 1 can be applied to *SpecificationIII*, we prove that the assumptions for Theorem 1 hold for *SpecificationIII* (analogously to Sections 4.3.1 and 4.4.1). The proof ideas are the same as for the corresponding proofs in Sections 4.3.1 and 4.4.1. Thus, we

do not describe the proof ideas again. The full proofs can be found in the appendix (see Sections A.3.20 to A.3.22).

We start by proving that the time bounds specified in module *SpecificationIII* meet Assumption 4.1:

Lemma 17. *In SpecificationIII it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T_d^y(s))$$

We prove Lemma 17 in the appendix in Section A.3.20. The proof is analogous to the proof of Lemma 4 that is sketched in Section 4.3.1.3. □

Lemma 18. *$relZTP^x$ of SpecificationIII^{time} meets Assumption 4.2, i.e. $\forall x \in \mathcal{X}, s \in \hat{\Sigma}$: $ZTP^x(s) \subseteq relZTP^x(s)$.*

We prove Lemma 18 in the appendix in Section A.3.21. The proof is analogous to the proof of Lemma 5 that is sketched in Section 4.3.1.3. □

The next assumption to show is that *SpecificationIII* fulfills Assumption 4.3, i.e. that for a $NextIII$ step from state s to state t it holds for all clocks $x \in \mathcal{X}$ that $relZTP^x(t) \subseteq relZTP^x(s)$.

Lemma 19. *For any two states s, t of SpecificationIII and all clocks $x \in \mathcal{X}$ (i.e., the clock LedgerTime), it holds that*

$$\textit{NextIII} \Rightarrow relZTP^{x\prime} \subseteq relZTP^x$$

We prove Lemma 19 in the appendix in Section A.3.22. The proof is analogous to the proof of Lemma 6 that is sketched in Section 4.3.1.3. □

Finally, we show that *SpecificationIII* implements *SpecificationIII^{time}*.

Theorem 6.

$$\textit{SpecificationIII} \Rightarrow \textit{SpecificationIII}^{time}$$

Proof: From Lemmas 17 to 19 it follows that *SpecificationIII* fulfills the assumptions Assumptions 4.1 to 4.3. Therefore, we can apply Theorem 1 to *SpecificationIII*. Because *SpecificationIII^{time}* is defined according to the definition of the time-optimized specification $Spec_{\hat{S}}$ in Section 4.1.3, this implies that *SpecificationIII* implements *SpecificationIII^{time}*. □

4.5.2. *SpecificationIII*^{time} Implements *SpecificationIV*

To show that *SpecificationIII*^{time} implements *SpecificationIV* we define a mapping f^{III} from *SpecificationIII*^{time} to *SpecificationIV* and we model check that the mapping f^{III} is a refinement mapping. The refinement mapping f^{III} is formally defined in TLA$^+$ in *SpecificationIII*^{time} in [GH26b]. The specification of the refinement mapping f^{III} is straightforward because *SpecificationIII*^{time} and *SpecificationIV* are defined similarly with the difference that *SpecificationIV* uses fewer variables (e.g., no *ChannelMessages[c]* or *UserPreimageInventory[u]*) and instead determines the possible states of payment based on the payment's state in other payment channels.

Lemma 20. *SpecificationIII*^{time} *is a refinement of SpecificationIV with the refinement mapping* f^{III}.

We check Lemma 20 using model checking (see Section 4.7). We model check *SpecificationIII*^{time} and verify that applying the mapping f^{III} to each step $s \rightarrow t$ of *SpecificationIII*^{time} leads to steps $f^{III}(s) \rightarrow f^{III}(t)$ of *SpecificationIV* and that each behavior of *SpecificationIII*^{time} is mapped to a behavior of *SpecificationIV*.

4.6. Abstraction of Payment Channel Network

Finally, we show that *SpecificationIV* using abstracted payment channels implements the abstract secure payment system defined in *SpecificationV*. We check this refinement by defining a refinement mapping and checking the refinement mapping using model checking (see Section 4.7). The definition of the refinement mapping is straightforward because all variables of *SpecificationV* are also variables of *SpecificationIV* and all variables are unchanged by the refinement mapping.

In the following section, we present our results of model checking. While the stepwise refinement approach significantly reduce the state space to explore, still only small finite models can be checked completely. To the extent that we could verify the refinements by model checking, we conclude from the refinement steps described above that *SpecificationI* implements *SpecificationV*, i.e. the specification of Lightning is an implementation of the abstract secure payment system and fulfills the security property.

4.7. Verification Results

In the previous sections, we have presented a stepwise refinement to show that the TLA$^+$ formalization of Lightning (*SpecificationI*) fulfills the security property (*SpecificationV*). We verify this stepwise refinement using proofs and model checking. In this section, we give an overview of the verification approaches and we present our results of model checking. We use exhaustive model checking to verify the four abstraction steps. For the first two

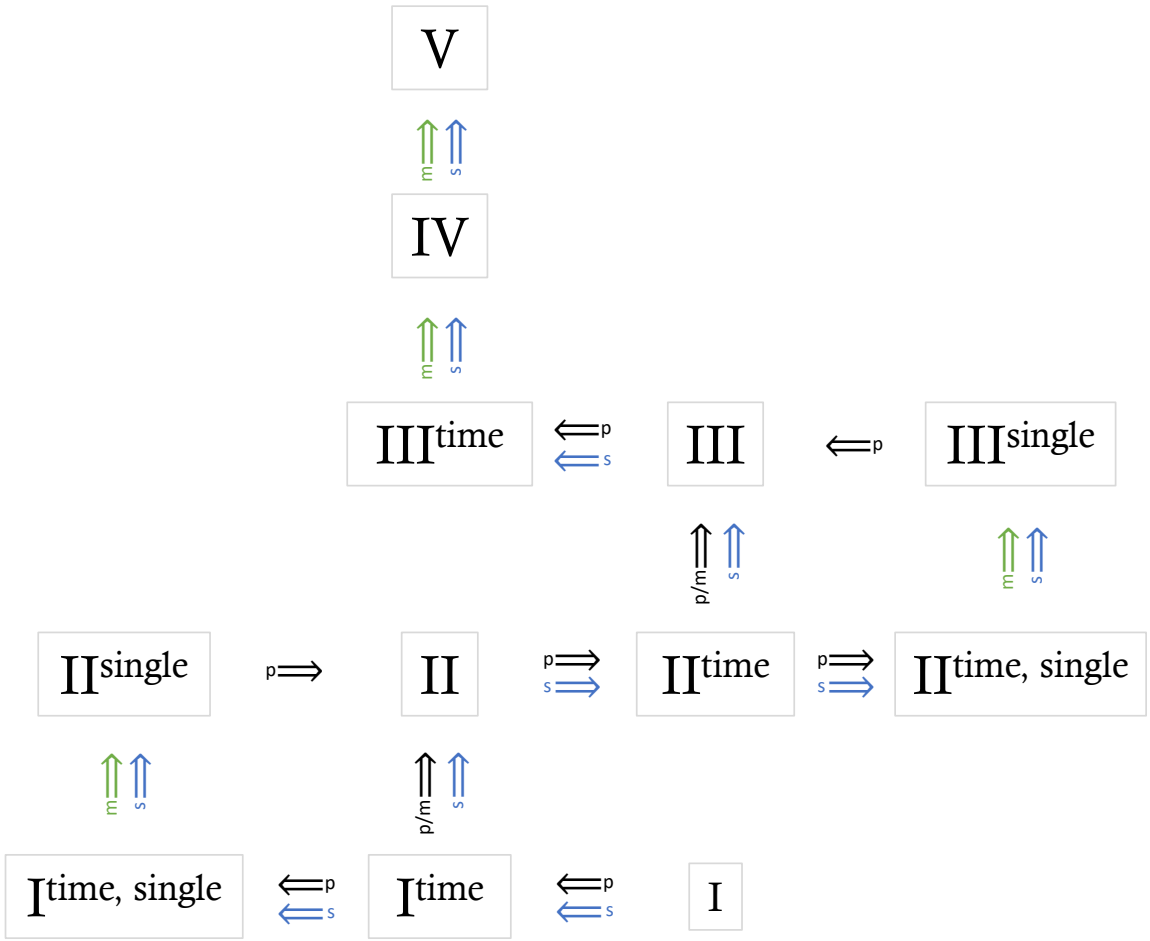

Figure 4.35.: Overview of the stepwise refinement showing the verification methods to verify each refinement. The Roman numerals represent the respective specifications. Black arrows labeled with 'p' denote refinements that have been verified by a proof, green arrows labeled with 'm' denote refinements that have been verified by exhaustive model checking, and blue arrows labeled with 's' denote refinements that have been verified by random simulation (partial model checking).

abstraction steps (*SpecificationI* to *SpecificationII* and *SpecificationII* to *SpecificationIII*) for which we use the decomposition approach, we model check the abstraction for a single payment channel. For the later two abstraction steps (*SpecificationII* to *SpecificationIV* and *SpecificationIV* to *SpecificationV*), we model check the abstraction for small payment channel networks. Besides using exhaustive model checking, we also use the model checking approach of random simulation in which random behaviors of a specification are generated and checked. We use this random simulation approach to verify the refinement steps with larger models that are too large to be exhaustively checked and we use random simulation as an additional check for the refinement steps that we have proven by pen-and-paper proofs.

Figure 4.35 gives an overview of the stepwise refinement and shows the verification methods that we use to verify each refinement. In the following, we give an overview of how we verify each step of the refinement. We have proven in Section 4.3.1 based on the proof

of Theorem 1 (see Section 4.1) that *SpecificationI* implements $SpecificationI^{time}$. We have shown the refinement from $SpecificationI^{time}$ to *SpecificationII* in Section 4.3.3 by a proof that is based on a lemma that is shown by model checking (see Section 4.3.3.2). This refinement step is marked in Fig. 4.35 by 'p/m' to indicate that this refinement was verified by a combination of proof and model checking. To prove the refinement from $SpecificationI^{time}$ to *SpecificationII*, we have proven that each payment channel in $SpecificationI^{time}$ is also modeled by $SpecificationI^{time,\ single}$. We used model checking to show that $SpecificationI^{time,\ single}$ refines $SpecificationII^{single}$. We further explain this model checking approach in Section 4.7.1. To complete the proof for the refinement from $SpecificationI^{time}$ to *SpecificationII*, we have proven that each payment channel specified by $SpecificationII^{single}$ can be combined with other channels in *SpecificationII*. From the results that *SpecificationI* implements $SpecificationI^{time}$ and that $SpecificationI^{time}$ implements *SpecificationII*, it follows by transitivity that *SpecificationI* implements *SpecificationII*. We have analogously shown that *SpecificationII* implements *SpecificationIII*. In a next step, we again applied Theorem 1 to show that *SpecificationIII* implements $SpecificationIII^{time}$. Finally, we model check that $SpecificationIII^{time}$ implements *SpecificationIV* and that *SpecificationIV* implements the security property *SpecificationV*. We also present these model checking results in Section 4.7.1. As an additional check, we use the model checking technique of random simulation for each refinement step except the steps from $SpecificationII^{single}$ to *SpecificationII* and from $SpecificationIII^{single}$ to *SpecificationIII*. We present the simulation approach in Section 4.7.2.

We use the explicit state model checker TLC [Var25b]. A model checking run works as follows: The model checker calculates all initial states by calculating all possible variable assignments that are allowed by the non-temporal part of the specification's definition (typically the $Init$ operator). The set of initial states is used to initialize the set of reachable states. From this set of reachable states, the model checker takes a state and calculates for each action of the specification whether the action is enabled by checking whether the unprimed formulas of the action are fulfilled for the state. If an action is enabled, the model checker uses the primed formulas of the action to calculate the next state and inserts the state in the set of reachable states. This procedure is repeated until all reachable states have been processed. The model checker terminates only if, at some point, no states are added to the set of reachable states because all states that are reached by steps of the specification are already in the set of reachable states.

To check that a specification S_1 fulfills another specification S_2, the model checker does not only explore the state space of specification S_1 but also performs a check when a new reachable state in specification S_1 is found. The model checker needs as input a refinement mapping from specification S_1 to specification S_2 that defines how a state of specification S_1 is mapped to a state of specification S_2. For each step that is found by the model checker from a state s to a state t, the model checker calculates the refinement mapping from specification S_1 to specification S_2 for both states s and t. Then, the model checker checks whether the resulting mapped states are allowed as a step by specification S_2. If the step is not allowed, a counterexample has been found and the model checker prints a trace of states that led to the violating step. If the specifications have liveness conditions, the model checker regularly verifies that each behavior of specification S_1 that is found is mapped by

the refinement mapping to a behavior of specification S_2 that fulfills the liveness condition of specification S_2.

With the stepwise refinement, we check that *SpecificationI*, the specification of Lightning, implements the security property *SpecificationV*. The security property would be fulfilled by a specification of a protocol that simply does nothing. To ensure that this is not the case for *SpecificationI*, we manually checked that there exist behaviors of *SpecificationI* in which payments are processed. We performed these checks by adding an invariant to our models that specifies that the state of a payment must not be PROCESSED. For each checked model, the model checker found a counterexample to this invariant which proves that there exist behaviors in which payments are processed.

In the remainder of this section, we present how we verified the refinement steps of the stepwise refinement using exhaustive model checking in Section 4.7.1 and random simulation in Section 4.7.2.

4.7.1. Verification By Exhaustive Model Checking

We check the most complex refinement steps using exhaustive model checking. Each behavior of the TLA$^+$ specification of Lightning or one of its abstractions starts with all users being in an initial state in which, for each channel to be opened, there is one user who has enough balance to fund the channel. While a behavior can terminate in the initial state, in a behavior of interest, at least one payment channel is opened. If one or multiple payment channels are open, users can send payments over the payment channels. At any point in time, a payment channel can be closed. As channels might be closed dishonestly, closing can be a complex process that involves multiple steps depending on the abstraction level in the stepwise refinement. A behavior can only terminate if all channels in which there is at least one honest user have been closed or have not been opened in the first place.

A *model* in the context of TLA$^+$ is a concrete instance of a specification in which all constants of the specification are defined by specific values. While our TLA$^+$ specifications describe infinitely many possible behaviors of a payment channel network, the models that we check define finite instances of specific payment channel networks and specific sets of payments that users want to process. For example, a model can be configured using a configuration as partially shown in Fig. 4.36. In Fig. 4.36, a payment channel network with four users and three channels is defined in which the user $UserB$ sends a payment of 5 coins over $UserC$ to $UserD$ and $UserC$ sends a payment of 2 coins to $UserB$.

We define several models that we use to verify the refinement steps. These models fall into three categories: Models for a single multi-hop payment, for two sequential payments in opposite directions, and for two concurrent payments in the same direction. By this selection of models, we cover the main challenges of payment channel networks while having models that are as small as possible: Using models with a single multi-hop payment we check that multi-hop payments are possible. For the refinement from *SpecificationIV* to *SpecificationV*, we can check models for a payment that involves up to six payment channels.

$$
\begin{aligned}
&UserInitialPayments \triangleq \langle \\
&\quad \{\}, \\
&\quad \{\ [id \mapsto 1, \\
&\qquad amount \mapsto 5, \\
&\qquad path \mapsto \langle [name \mapsto \text{“UserB”}],\ [name \mapsto \text{“UserC”}],\ [name \mapsto \text{“UserD”}] \rangle, \\
&\qquad absTimelock \mapsto 12]\}, \\
&\quad \{\ [id \mapsto 2, \\
&\qquad amount \mapsto 2, \\
&\qquad path \mapsto \langle [name \mapsto \text{“UserC”}],\ [name \mapsto \text{“UserB”}] \rangle, \\
&\qquad absTimelock \mapsto 16]\}, \\
&\quad \{\} \\
&\rangle \\
&NameForUserID \triangleq \langle \\
&\quad [name \mapsto \text{“UserA”}], \\
&\quad [name \mapsto \text{“UserB”}], \\
&\quad [name \mapsto \text{“UserC”}], \\
&\quad [name \mapsto \text{“UserD”}] \\
&\rangle \\
&UserChannelFundingAmount \triangleq \langle \\
&\quad 1 :> 10, \\
&\quad 2 :> 10, \\
&\quad 3 :> 10, \\
&\quad 4 :> 0 \\
&\rangle
\end{aligned}
$$

Figure 4.36.: Excerpt of the configuration of an exemplary model. The configuration describes a payment channel network with four users and three channels. It is implicitly assumed that between each pair of users with successive user ids a channel will be opened. The first channel is funded by $UserA$ with 10 coins, the second channel is funded by $UserB$ with 10 coins, and the third channel is funded by $UserC$ with 10 coins. The configuration defines the values of three constants: *UserInitialPayments* defines a sequence that contains for each user the payments that the user should send. The first entry in the sequence defines the set of payments sent by the user with id 1, the second entry for the user with id 2, etc. For each payment, an id, the payment's amount, the payment's route, and an absolute timelock is defined. Similarly, *NameForUserID* assigns a name to each user id. These names are used, for example, in the specification of a payment's route. The constant *UserChannelFundingAmount* defines for each user an assignment of a channel id to the amount that the user uses to fund the channel. The notation $1 :> 10$ has the meaning that the channel with id 1 is funded with 10 coins.

Models that contain two payments are relevant to check because payment channels are meant to be used for multiple payments. We check models with two payments in opposite directions. Because payment channels are single-funded, directly after opening a payment channel, payments can only be sent in one direction. Thus, the payment in the opposite direction becomes possible only after the first payment was successfully processed. Such models are relevant to check because the payments in these models affect each other because one payment is dependent on the other payment. We further check models with two concurrent payments in the same direction. These models are relevant because there are many possibilities of how the two payments can interleave and some steps might

Table 4.4.: Model checking results for the refinement mapping from *SpecificationIV* to *SpecificationV* for a single payment

Description	Payment Flow	# States	Runtime
One hop	A → B	10^2	< 1 min
Two hops	A → B → C	10^3	< 1 min
Three hops	A → B → C → D	10^4	< 1 min
Four hops	A → B → C → D → E	10^5	~ 1 min
Five hops	A → B → C → D → E → F	10^6	~ 30 min
Six hops	A → B → C → D → E → F → G	10^7	~ 12 h

influence both payments at the same time. However, because of the many interleavings, these models lead to a vast state space especially in the low-level specification of Lightning so that we can not completely model check as many models with parallel payments for the refinement from $SpecificationI^{time,\ single}$ to *SpecificationII* as we can for the other refinement steps. We argue that models incorporating more than two payments do not generate a sufficient number of novel, relevant behavioral patterns to justify the substantially increased complexity associated with their significantly larger state space. Therefore, we check such models only with random simulation (see Section 4.7.2).

All models have in common that a payment channel network with adjacent channels is modeled, i.e., the network is a single component and each user has a degree of at most 2. While modeling networks with branches would be possible, such networks are not expected to lead to different results because multi-hop payments in Lightning are made through a chain of hops from the sender to the receiver and cannot be branched. For each refinement, we check models in which a single payment is made and models in which two payments are made.

For each checked refinement, each model is listed in a table that shows the model's number of distinct states in the state space rounded to the nearest multiple of a power of 10. The tables also show the approximate runtime for checking the model with the model checker TLC on a server with an AMD EPYC 9654 96-Core Processor where the model checker TLC runs with 96 workers and is allowed to use 86 GB of RAM. The runtimes are from a single run and are meant only to give an impression. The runtimes might vary significantly if the models are checked on other hardware.

In the following, we first discuss in Section 4.7.1.1 our results for models of single payments. Then, we present in Section 4.7.1.2 our results of models with two sequential payments. Finally, we show the results of models with two parallel payments in Section 4.7.1.3.

4.7.1.1. Models With One Multi-Hop Payment

As first category of models used for model checking, we define models with a payment channel network and a single payment that is sent over multiple hops. The models vary by the number of hops of the multi-hop payments. The payment channel network in each

Table 4.5.: Model checking results for the refinement mapping from $SpecificationIII^{time}$ to *SpecificationIV* for a single payment

Description	Payment Flow	# States	Runtime
One hop	A → B	6×10^2	< 1 min
Two hops	A → B → C	1×10^4	< 1 min
Three hops	A → B → C → D	2×10^5	~ 1 min
Four hops	A → B → C → D → E	3×10^6	~ 45 min
Five hops	A → B → C → D → E → F	6×10^7	~ 23 h

Table 4.6.: Model checking results for the refinement mapping from $SpecificationII^{time,\ single}$ to $SpecificationIII^{single}$ for a single payment

Description	Payment Flow	# States	Runtime
Direct payment	A → B	2×10^3	< 1 min
Direct payment of full balance in channel	A → B	2×10^3	< 1 min
Direct payment in reverse direction	A ← B	2×10^3	< 1 min
Forwarded by A	→ A → B	3×10^3	< 1 min
Forwarded by B	A → B →	1×10^4	< 1 min
Forwarded by A and B	→ A → B →	1×10^4	< 1 min

model is only as large as required for the multi-hop payment, i.e. for a multi-hop payment over two hops, the payment channel network consists of two payment channels and three users. Table 4.4 lists the models and the results for the refinement from *SpecificationIV* to *SpecificationV*. To facilitate describing the models, we assume that the users in the network have names from A to G and each pair of adjacent users in alphabetical order is connected with a payment channel. In the tables that present the results of checking specific models, we represent the models with the following notation that models the payment flow. An arrow from one user to another user, e.g., A → B, means that user A sends a payment to user B. If the payment is forwarded by user B to user C, then the complete multi-hop payment over two channels is shown as A → B → C. Table 4.4 shows models for a single-hop payment up to a multi-hop payment over six hops. It can be seen

Table 4.7.: Model checking results for the refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$ for a single payment

Description	Payment Flow	# States	Runtime
Direct payment	A → B	8×10^4	~ 1 min
Direct payment of full balance in channel	A → B	7×10^4	~ 1 min
Direct payment in reverse direction	A ← B	8×10^4	~ 1 min
Forwarded by A	→ A → B	1×10^5	~ 3 min
Forwarded by B	A → B →	3×10^5	~ 5 min
Forwarded by A and B	→ A → B →	3×10^5	~ 5 min

that the number of states and the necessary time to check each model increases by an order of magnitude for each additional hop.

We use the same models to model check the refinement mapping from $SpecificationIII^{time}$ to *SpecificationIV*. The results are listed in Table 4.5. Again, it can be seen that the state space increases with the number of hops. A comparison with the results for the refinement from *SpecificationIV* to *SpecificationV* shows that for the refinement from $SpecificationIII^{time}$ to *SpecificationIV* the state space increases by a larger factor. This is due to the higher complexity of $SpecificationIII^{time}$ compared to *SpecificationIV* and, thus, a higher number of steps that need to be checked for each additional channel. Because of the models' higher runtimes, we only check models up to five hops for the refinement from $SpecificationIII^{time}$ to *SpecificationIV*.

We cannot directly use the same models to check the refinements between $SpecificationI^{time,\ single}$ and $SpecificationII^{single}$ and between $SpecificationII^{time,\ single}$ and $SpecificationIII^{single}$ because these specifications are for a single channel only with two users. Thus, models with more than one payment channel and more than two users cannot be checked. Instead, we check models that represent every scenario in which single payment channel can be during a multi-hop payment: The simplest case is a single hop payment between two users. For actual multi-hop payments, a channel can either be the channel in which the payment is sent by the sender, a channel that forwards the payment, or the channel in which the payment is received by the receiver. We check models for these four scenarios for the refinements between $SpecificationI^{time,\ single}$ and $SpecificationII^{single}$ and between $SpecificationII^{time,\ single}$ and $SpecificationIII^{single}$. The results are listed in Tables 4.6 and 4.7. In the tables, we represent that a user A forwards a payment that user A has received from another payment channel in its environment by drawing an arrow that does not start at another user, e.g., $\rightarrow$ A $\rightarrow$ B represents a payment that is forwarded by user A and received by user B. In addition to the models that we have already mentioned, there is also a model to check the edge case that the funder sends the entire balance of the channel to the other user and there is a model in which the roles of both users A and B are reversed to check that the specification does not statically assume that a channel is funded by user A but that a channel can also be funded by user B.

As shown in Tables 4.6 and 4.7, the models for a single payment are fully explored within a few minutes for $SpecificationI^{time,\ single}$ and below a minute for $SpecificationII^{time,\ single}$. For these models, the state space of $SpecificationI^{time,\ single}$ (see Table 4.7) is about an order of magnitude larger than the state space of $SpecificationII^{time,\ single}$ (see Table 4.6). This difference indicates the large gap in abstraction between these two specifications. While $SpecificationI^{time,\ single}$ specifies all detailed steps of each user to open, update, and close a payment channel, many of these steps are abstracted by a single step of $SpecificationII^{single}$.

Putting together the model checking results for one multi-hop payment, we conclude that we have a complete chain of successful model checking and proofs from *SpecificationI* to *SpecificationV* that show that a multi-hop payment with up to five hops in the TLA$^+$ specification of Lightning fulfills the security property. Because the models for the specifications of a single channel cover all possible options in which a channel can be during

a multi-hop payment, the proofs for the refinements of *SpecificationI* to *SpecificationII* and from *SpecificationII* to *SpecificationIII* extend these results for a single channel to a channel network in which each channel performs at most one payment. Because we have checked the refinements of $SpecificationIII^{time}$ to *SpecificationIV* and from *SpecificationIV* to *SpecificationV* for multi-hop payments with up to five hops, we conclude that behaviors with at most one multi-hop payment are secure. Because the idea of payment channels is, however, to make multiple payments, we describe in the following our approach to check models with multiple payments.

4.7.1.2. Models With a Two Sequential Multi-Hop Payments

To check the effects that payments can have on each other, we further check models with two payments. The first group of models with two payments are two payments in opposite directions. Because channels are single-funded, initially only one payment can be processed. After the first payment has been processed, the payment in the opposite direction can be processed. The second group of models with two payments are two payments in the same direction that we discuss in Section 4.7.1.3. We start with the first group of payments in opposite directions.

In contrast to the presentation of a single payment above, we start by presenting the results for the lowest-level specifications, i.e. the refinement from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$. As for the single payment, we aim to create models that cover all possible situations in which a payment channel can be in a network where two payments are sent in opposite directions. For each of the two payments, each user can be either the sender or the receiver of the payment (depending on the direction of the payment) or an intermediate node that forwards the payment. Thus, there are 16 different options for a payment channel that processes both payments: There are two users and two payments. Each user can either be sender/receiver or intermediate. Thus, there are four options for each payment. Consequently, for two payments, there are $4 \times 4 = 16$ options. These 16 models are listed in Table 4.8 along with the results of checking each model for the refinement from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$. These models are significantly larger than the models for a single payment because longer behaviors are possible and, in particular, there are more possibilities for dishonest behavior because the channel is updated more often and, thus, there are more outdated transactions. Table 4.9 shows that the same models can be checked for the refinement from $SpecificationII^{time,\ single}$ to $SpecificationIII^{single}$ in significantly lower time.

In a similar way, we check all 16 options for a single channel with two payments also for the higher-level specifications $SpecificationIII^{time}$ and *SpecificationIV*. To this end, we consider payment channel networks with up to three payment channels so that the middle channel is checked in all 16 options. Tables 4.10 and 4.11 show the model checking results for the refinements from $SpecificationIII^{time}$ to *SpecificationIV* and from *SpecificationIV* to *SpecificationV*. The models in which the two payments are processed by all three payment channels have a large state space even for $SpecificationIII^{time}$ because the state space increases exponentially with the number of channels and with the number of payments

Table 4.8.: Model checking results for the refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$ for two payments in opposite directions

Description	Payment Flow	# States	Runtime
Both payments direct	B → C B ← C	1×10^6	~ 30 min
First payment forwarded by C, second payment direct	B → C → B ← C	3×10^6	~ 1 h
Both payments forwarded by C	B → C → B ← C ←	5×10^6	~ 2 h
First payment direct, second payment forwarded by C	B → C B ← C ←	2×10^6	~ 50 min
First payment forwarded by B, second payment direct	→ B → C B ← C	2×10^6	~ 80 min
First payment forwarded by B and C, second payment direct	→ B → C → B ← C	3×10^6	~ 90 min
First payment forwarded by B and C, second payment forwarded by C	→ B → C → B ← C ←	4×10^6	~ 2 h
First payment forwarded by B, second payment forwarded by C	→ B → C B ← C ←	4×10^6	~ 2 h
First payment direct, second payment forwarded by B	B → C ← B ← C	2×10^6	~ 1 h
First payment forwarded by C, second payment forwarded by B	B → C → ← B ← C	6×10^6	~ 2 h
First payment forwarded by C, second payment forwarded by B and C	B → C → ← B ← C ←	5×10^6	~ 2 h
First payment direct, second payment forwarded by B and C	B → C ← B ← C ←	2×10^6	~ 1 h
Both payments forwarded by B	→ B → C ← B ← C	5×10^6	~ 3 h
First payment forwarded by B and C, second payment forwarded by B	→ B → C → ← B ← C	6×10^6	~ 3 h
Both payments forwarded by B and C	→ B → C → ← B ← C ←	5×10^6	~ 2 h
First payment forwarded by B, second payment forwarded by B and C	→ B → C ← B ← C ←	4×10^6	~ 2 h

Table 4.9.: Model checking results for the refinement mapping from $SpecificationII^{time,\ single}$ to $SpecificationIII^{single}$ for two payments in opposite directions

Description	Payment Flow	# States	Runtime
Both payments direct	B → C B ← C	1×10^4	< 1 min
First payment forwarded by C, second payment direct	B → C → B ← C	5×10^4	< 1 min
Both payments forwarded by C	B → C → B ← C ←	9×10^4	< 1 min
First payment direct, second payment forwarded by C	B → C B ← C ←	2×10^4	< 1 min
First payment forwarded by B, second payment direct	→ B → C B ← C	3×10^4	< 1 min
First payment forwarded by B and C, second payment direct	→ B → C → B ← C	7×10^4	< 1 min
First payment forwarded by B and C, second payment forwarded by C	→ B → C → B ← C ←	8×10^4	< 1 min
First payment forwarded by B, second payment forwarded by C	→ B → C B ← C ←	4×10^4	< 1 min
First payment direct, second payment forwarded by B	B → C ← B ← C	4×10^4	< 1 min
First payment forwarded by C, second payment forwarded by B	B → C → ← B ← C	1×10^5	< 1 min
First payment forwarded by C, second payment forwarded by B and C	B → C → ← B ← C ←	1×10^5	< 1 min
First payment direct, second payment forwarded by B and C	B → C ← B ← C ←	4×10^4	< 1 min
Both payments forwarded by B	→ B → C ← B ← C	8×10^4	< 1 min
First payment forwarded by B and C, second payment forwarded by B	→ B → C → ← B ← C	1×10^5	< 1 min
Both payments forwarded by B and C	→ B → C → ← B ← C ←	8×10^4	< 1 min
First payment forwarded by B, second payment forwarded by B and C	→ B → C ← B ← C ←	7×10^4	< 1 min

Table 4.10.: Model checking results for the refinement mapping from *SpecificationIII*^{time} to *SpecificationIV* for two payments in opposite directions

Description	Payment Flow	# States	Runtime
Both payments direct	$B \rightarrow C$ $B \leftarrow C$	9×10^3	< 1 min
First payment forwarded by C, second payment direct	$B \rightarrow C \rightarrow D$ $B \leftarrow C$	2×10^5	~ 1 min
Both payments forwarded by C	$B \rightarrow C \rightarrow D$ $B \leftarrow C \leftarrow D$	4×10^5	~ 4 min
First payment direct, second payment forwarded by C	$B \rightarrow C$ $B \leftarrow C \leftarrow D$	2×10^5	~ 1 min
First payment forwarded by B, second payment direct	$A \rightarrow B \rightarrow C$ $B \leftarrow C$	2×10^5	< 1 min
First payment forwarded by B and C, second payment direct	$A \rightarrow B \rightarrow C \rightarrow D$ $B \leftarrow C$	3×10^6	~ 30 min
First payment forwarded by B and C, second payment forwarded by C	$A \rightarrow B \rightarrow C \rightarrow D$ $B \leftarrow C \leftarrow D$	8×10^6	~ 2 h
First payment forwarded by B, second payment forwarded by C	$A \rightarrow B \rightarrow C$ $B \leftarrow C \leftarrow D$	3×10^6	~ 30 min
First payment direct, second payment forwarded by B	$B \rightarrow C$ $A \leftarrow B \leftarrow C$	2×10^5	~ 1 min
First payment forwarded by C, second payment forwarded by B	$B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C$	2×10^6	~ 25 min
First payment forwarded by C, second payment forwarded by B and C	$B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C \leftarrow D$	5×10^6	~ 90 min
First payment direct, second payment forwarded by B and C	$B \rightarrow C$ $A \leftarrow B \leftarrow C \leftarrow D$	2×10^6	~ 25 min
Both payments forwarded by B	$A \rightarrow B \rightarrow C$ $A \leftarrow B \leftarrow C$	4×10^5	~ 5 min
First payment forwarded by B and C, second payment forwarded by B	$A \rightarrow B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C$	7×10^6	~ 2 h
Both payments forwarded by B and C	$A \rightarrow B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C \leftarrow D$	2×10^7	~ 7 h
First payment forwarded by B, second payment forwarded by B and C	$A \rightarrow B \rightarrow C$ $A \leftarrow B \leftarrow C \leftarrow D$	7×10^6	~ 2 h

Table 4.11.: Model checking results for the refinement mapping from *SpecificationIV* to *SpecificationV* for two payments in opposite directions

Description	Payment Flow	# States	Runtime
Both payments direct	$B \rightarrow C$ $B \leftarrow C$	5×10^2	< 1 min
First payment forwarded by C, second payment direct	$B \rightarrow C \rightarrow D$ $B \leftarrow C$	6×10^3	< 1 min
Both payments forwarded by C	$B \rightarrow C \rightarrow D$ $B \leftarrow C \leftarrow D$	9×10^3	< 1 min
First payment direct, second payment forwarded by C	$B \rightarrow C$ $B \leftarrow C \leftarrow D$	6×10^3	< 1 min
First payment forwarded by B, second payment direct	$A \rightarrow B \rightarrow C$ $B \leftarrow C$	6×10^3	< 1 min
First payment forwarded by B and C, second payment direct	$A \rightarrow B \rightarrow C \rightarrow D$ $B \leftarrow C$	7×10^4	< 1 min
First payment forwarded by B and C, second payment forwarded by C	$A \rightarrow B \rightarrow C \rightarrow D$ $B \leftarrow C \leftarrow D$	1×10^5	< 1 min
First payment forwarded by B, second payment forwarded by C	$A \rightarrow B \rightarrow C$ $B \leftarrow C \leftarrow D$	7×10^4	< 1 min
First payment direct, second payment forwarded by B	$B \rightarrow C$ $A \leftarrow B \leftarrow C$	5×10^3	< 1 min
First payment forwarded by C, second payment forwarded by B	$B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C$	6×10^4	< 1 min
First payment forwarded by C, second payment forwarded by B and C	$B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C \leftarrow D$	8×10^4	< 1 min
First payment direct, second payment forwarded by B and C	$B \rightarrow C$ $A \leftarrow B \leftarrow C \leftarrow D$	5×10^4	< 1 min
Both payments forwarded by B	$A \rightarrow B \rightarrow C$ $A \leftarrow B \leftarrow C$	9×10^3	< 1 min
First payment forwarded by B and C, second payment forwarded by B	$A \rightarrow B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C$	1×10^5	~ 1 min
Both payments forwarded by B and C	$A \rightarrow B \rightarrow C \rightarrow D$ $A \leftarrow B \leftarrow C \leftarrow D$	2×10^5	~ 2 min
First payment forwarded by B, second payment forwarded by B and C	$A \rightarrow B \rightarrow C$ $A \leftarrow B \leftarrow C \leftarrow D$	1×10^5	< 1 min

Table 4.12.: Model checking results for the refinement mapping from $SpecificationI^{time,\ single}$ to $SpecificationII^{single}$ for two payments in the same direction

Description	Payment Flow	# States	Runtime
Both payments direct	B → C B → C	6×10^7	~ 19 h
One payment forwarded by C, the other payment direct	B → C → B → C	2×10^8	~ 3 d
Both payments forwarded by C	B → C → B → C →	1×10^9	> 16 d 89 %
One payment forwarded by B, the other payment direct	→ B → C B → C	3×10^8	~ 4 d
One payment forwarded by B and C, the other payment direct	→ B → C → B → C	7×10^8	~ 9 d
One payment forwarded by B and C, the other payment forwarded by C	→ B → C → B → C →	3×10^9	> 16 d 41 %
One payment forwarded by B, the other payment forwarded by C	→ B → C B → C →	1×10^9	> 12 d 85 %
Both payments forwarded by B	→ B → C → B → C	5×10^8	~ 7 d
One payment forwarded by B and C, the other payment forwarded by B	→ B → C → → B → C	9×10^8	> 12 d 94 %
Both payments forwarded by B and C	→ B → C → → B → C →	3×10^9	> 19 d 51 %

processed by each channel. The runtimes shown in Table 4.11 for *SpecificationIV* show that the abstraction from *SpecificationIII* to *SpecificationIV* significantly reduces the state space by omitting modeling of time and preimages.

4.7.1.3. Models With a Two Concurrent Multi-Hop Payments

The second group of models with two payments are models with two payments in the same direction. Because these payments can be processed concurrently, the models for payments in the same direction have a larger state space than the models for sequential payments. We again consider all possible options for two parallel payments in a payment channel. Because we do not distinguish which payment goes first, there are only ten unique options compared to the 16 options for sequential payments. For $SpecificationI^{time,\ single}$, the state space is so large that we completely model check only five of the ten models and model check

Table 4.13.: Model checking results for the refinement mapping from $SpecificationII^{time,\ single}$ to $SpecificationIII^{single}$ for two payments in the same direction

Description	Payment Flow	# States	Runtime
Both payments direct	B → C B → C	5×10^4	< 1 min
One payment forwarded by C, the other payment direct	B → C → B → C	2×10^5	~ 1 min
Both payments forwarded by C	B → C → B → C →	2×10^6	~ 10 min
One payment forwarded by B, the other payment direct	→ B → C B → C	1×10^5	< 1 min
One payment forwarded by B and C, the other payment direct	→ B → C → B → C	4×10^5	~ 2 min
One payment forwarded by B and C, the other payment forwarded by C	→ B → C → B → C →	2×10^6	~ 14 min
One payment forwarded by B, the other payment forwarded by C	→ B → C B → C →	6×10^5	~ 3 min
Both payments forwarded by B	→ B → C → B → C	2×10^5	~ 1 min
One payment forwarded by B and C, the other payment forwarded by B	→ B → C → → B → C	6×10^5	~ 2 min
Both payments forwarded by B and C	→ B → C → → B → C →	2×10^6	~ 14 min

the remaining models only partially. The models are listed in Table 4.12. For the models for which we have only partial model checking results, the right column of Table 4.12 shows the time that the model checker spent checking the model and the share of the state space that was explored during that time. E.g, for the last model in Table 4.12, the model checker needed 19 days to check 51 % of the model's state space which has a size of about 3×10^9 distinct states. For the refinement from $SpecificationII^{time,\ single}$ to *SpecificationIII*, we can check all models as shown in Table 4.13. Tables 4.12 and 4.13 show that the state space of the models for two concurrent payments is significantly larger than the state space for the corresponding models with two sequential payments shown in Tables 4.8 and 4.9. Tables 4.14 and 4.15 show the results for the refinements from $SpecificationIII^{time}$ to *SpecificationIV* and from *SpecificationIV* to *SpecificationV*. Analogously to the models for sequential payments, these models specify payment channel networks between one channel and three channels depending on the number of channels involved in the two

Table 4.14.: Model checking results for the refinement mapping from *SpecificationIII*^{time} to *SpecificationIV* for two payments in the same direction

Description	Payment Flow	# States	Runtime
Both payments direct	$B \to C$ $B \to C$	1×10^4	< 1 min
One payment forwarded by C, the other payment direct	$B \to C \to D$ $B \to C$	2×10^5	~ 1 min
Both payments forwarded by C	$B \to C \to D$ $B \to C \to D$	6×10^5	~ 7 min
One payment forwarded by B, the other payment direct	$A \to B \to C$ $B \to C$	2×10^5	~ 1 min
One payment forwarded by B and C, the other payment direct	$A \to B \to C \to D$ $B \to C$	4×10^6	~ 45 min
One payment forwarded by B and C, the other payment forwarded by C	$A \to B \to C \to D$ $B \to C \to D$	1×10^7	~ 3 h
One payment forwarded by B, the other payment forwarded by C	$A \to B \to C$ $B \to C \to D$	4×10^6	~ 40 min
Both payments forwarded by B	$A \to B \to C$ $A \to B \to C$	6×10^5	~ 9 min
One payment forwarded by B and C, the other payment forwarded by B	$A \to B \to C \to D$ $A \to B \to C$	1×10^7	~ 3 h
Both payments forwarded by B and C	$A \to B \to C \to D$ $A \to B \to C \to D$	4×10^7	~ 19 h

payments. Again, it can be seen that the abstraction from *SpecificationIII* to *SpecificationIV* significantly reduces the state space.

4.7.2. Verification By Random Simulation

We could exhaustively model check most models for one payment and two payments for all four abstractions. Therefore, we have high confidence that there is no behavior of Lightning that breaks the security property with one or two payments. While most problems would probably manifest already with two payments, there might be a problem in the protocol that is not found by the model checking approach as described above. To reduce the probability of not knowing such a behavior, we use random simulation to explore and check larger models that we cannot model check exhaustively. We use

Table 4.15.: Model checking results for the refinement mapping from *SpecificationIV* to *SpecificationV* for two payments in the same direction

Description	Payment Flow	# States	Runtime
Both payments direct	$B \to C$ $B \to C$	6×10^2	< 1 min
One payment forwarded by C, the other payment direct	$B \to C \to D$ $B \to C$	8×10^3	< 1 min
Both payments forwarded by C	$B \to C \to D$ $B \to C \to D$	1×10^4	< 1 min
One payment forwarded by B, the other payment direct	$A \to B \to C$ $B \to C$	8×10^3	< 1 min
One payment forwarded by B and C, the other payment direct	$A \to B \to C \to D$ $B \to C$	1×10^5	< 1 min
One payment forwarded by B and C, the other payment forwarded by C	$A \to B \to C \to D$ $B \to C \to D$	2×10^5	~ 2 min
One payment forwarded by B, the other payment forwarded by C	$A \to B \to C$ $B \to C \to D$	1×10^5	< 1 min
Both payments forwarded by B	$A \to B \to C$ $A \to B \to C$	1×10^4	< 1 min
One payment forwarded by B and C, the other payment forwarded by B	$A \to B \to C \to D$ $A \to B \to C$	2×10^5	~ 3 min
Both payments forwarded by B and C	$A \to B \to C \to D$ $A \to B \to C \to D$	4×10^5	~ 8 min

random simulation to check models with more than two payments for all refinement steps as shown in Fig. 4.35.

During random simulation, the model checker generates random behaviors and checks whether a property, in our case, a refinement, is fulfilled by the behavior. The model checker generates a random behavior by starting in a randomly chosen initial state and finding all possible successors of this state. From all possible successor states, one state is chosen uniformly at random. Then, the process is repeated by finding all possible successors of the new state. Thereby, the model checker finds a random behavior of the system.

When performing random simulation, there is a challenge to ensure that the model checker actually explores behaviors of interest, in particular, behaviors that are long enough so that the system can make progress. In the TLA$^+$ specification of Lightning and in the abstractions, after a channel has been opened, there exists in every state at

least one successor state for closing the channel. Because the model checker chooses each successor state uniformly at random and there might only be a few successor states that model progress of the protocol, the model checker tends to create behaviors in which a channel is closed shortly after being opened. We approach this challenge by modifying the specifications specifically for random simulation: We introduce a constant *MIN_SIMULATION_LENGTH* to specify that the model checker may only take a step that closes a channel if the current length of the behavior created by the model checker is at least *MIN_SIMULATION_LENGTH*. Further, we require that, while the current length of the behavior is below *MIN_SIMULATION_LENGTH*, even dishonest users are active and may not prevent the protocol from making progress by stalling. An alternative approach used by concurrent work [How+25] would be to manually weight actions that do not make progress to reduce the likelihood of them being chosen as successor states.

We run random simulation for all refinement steps as shown in Fig. 4.35. During random simulation, the model checker TLC creates behaviors of up to 300 states in length. We run random simulation for each refinement step for 20 values for the constant *MIN_SIMULATION_LENGTH* between 0 and 280. The values are chosen depending on the refinement step because for higher abstracted specifications, fewer steps need to be taken even for behaviors with multiple payments. For each of the 20 values, we run the random simulation for 20 minutes. Thus, each refinement step is checked by random simulation for about six hours.

While random simulation helped us to quickly catch errors in drafts of our specifications and refinement mappings, for the current version of the specifications even repeated execution of random simulation has not found any errors.

4.8. Discussion

In this section, we conclude the chapter by discussing our method and future work. We start by reviewing in Section 4.8.1 our choice to model Lightning in TLA$^+$. In Section 4.8.2, we discuss limitations of our specification of Lightning and potential future work of extending the formalization. While we conclude that Lightning fulfills the security property, several attacks on Lightning have been presented. In Section 4.8.3 we categorize these attacks and explain why the existence of these attacks does not contradict our results. We discuss in Section 4.8.4 a use case of using the TLA$^+$ formalization of Lightning for evaluating new variants of Lightning. Because to be secure in practice, it does not suffice that the implemented protocol has a secure design but it needs to be ensured that the implementation is actually secure, we discuss in Section 4.8.5 an approach to use the formalization of Lightning in TLA$^+$ to check the security of executable implementations of Lightning.

4.8.1. Choice of Formalization Language and Tools

Protocol verifiers such as Tamarin [Mei+13] and ProVerif [Bla16] have been used successfully for verification of numerous security protocols [Bas+22]. These verifiers support unbounded verification, i.e., they can reason about a protocol's properties without the limitations of finite model checking. Prior work [BGW20; HS22; HS20] has used Tamarin and ProVerif to model and verify selected aspects of Lightning. A key advantage of these verifies is that they support modeling cryptographic primitives and allow for stronger adversary models. However, the verification approaches employed by Tamarin and ProVerif, make it challenging to model natural numbers with addition and subtraction of variables (see [Tea24b, page 35] and [Bla22, page 18]). This limitation is particularly problematic for reasoning about blockchain transactions with amounts. In contrast, while using TLA$^+$ for a formalization requires abstraction of cryptographic primitives, it enables explicit modeling and verification of numerical properties. As a result, we are able to model and verify key aspects of Lightning most notably the correctness of users' balances.

Previous and concurrent work that considered the security of Lightning used the Universal Composability (UC) framework [Can01] to verify Lightning [KT20] and the program verification platform Why3 [Bob+11] to verify a simplified variant of Lightning [FSL26]. Both approaches allow for modeling stronger adversaries than in our approach. Verification in the UC framework comes with the challenge that complex proofs cannot be automatically verified and are, thus, prone to undetected errors. Working on the verification of the TLA$^+$ specification of Lightning, we found two flaws in the UC formalization of Lightning and the associated proof (see Section 3.1.1). With the verification platform Why3, machine-checked proofs can be written. The work of Fabiański et al. [FSL26] shows, however, that this requires a large effort even for a variant of Lightning without HTLCs and multi-hop payments.

A benefit of TLA$^+$ is that TLA$^+$ offers multiple ways to prove properties: manually, using an explicit-state [Var25b] or a symbolic model checker [Var25a], or a tool for automated reasoning [Cen22]. We used the explicit-state model checker TLC for checking the steps of the stepwise refinement. The automated model checking process and the generation of counterexamples facilitated our process of defining the intermediate specifications and the associated refinement mappings. However, by using model checking we could only check models for relatively small scenarios with multi-hop payments over up to six hops and up to two sequential or parallel payments. Theoretically, there might be attacks that only apply when there are more than two concurrent payments in a payment channel; these attacks would not be discovered by our approach.

Further, to use model checking, we had to restrict the adversary model so that the adversary has a limited scope of possible actions; still, our adversary model describes the relevant adversarial actions (see Section 3.3.5.3). A theorem prover [Cen22] could be used to verify the TLA$^+$ specification of Lightning with an even stronger adversary model. Currently, writing such a proof seems to be too effortful. However, it will be facilitated by future advancements in assistance and automation for theorem proving, possibly by the use of large language models [ZT26].

4.8.2. Limitations and Future Work

While our work represents a step toward a comprehensive formal verification of Lightning, it has limitations both with respect to the verification methodology, as discussed above, and the scope of the protocol model itself. To make the formalization tractable, we abstracted away protocol aspects that are not essential for reasoning about security properties. These include, for example, routing fees paid by the sender of a payment to intermediate hops, key rotation and onion-routing for enhanced privacy, and route discovery for multi-hop payments. Moreover, the blockchain model could be extended to better reflect real-world conditions by incorporating transaction fees and blockchain reorganizations. In Chapter 5, we show how the blockchain model can be adapted to include transaction confirmation delays.

We further assume a static set of participants: all parties are known at initialization, channels are opened once and eventually closed. The formalization could be extended to model scenarios in which channels are reopened or new participants dynamically join the network.

Our adversary model restricts the capabilities of an adversary by disallowing adversarial users to send messages with arbitrary content and to exchange information with other adversarial users. While these constraints were necessary to keep the state space of the specification manageable, future work could explore optimizations that enable the analysis of stronger and more expressive adversary models without compromising tractability.

During the course of this work, the official Lightning specification was extended to support dual-funded channels, allowing both channel participants to deposit coins during channel opening (see [Lig26c, BOLT #2, Channel Establishment v2]). The methodology introduced in this work can be directly applied to formalize and analyze the security implications of this extension.

Finally, the property verified in this work focuses primarily on security. An interesting direction for future research is the inclusion of additional properties in the abstract secure payment system and corresponding adaptations of the verification approach. For example, it could be shown that, under assumptions of honest and cooperative participants and timely message delivery, payments are guaranteed to succeed.

4.8.3. Known Attacks on Lightning

While we prove that Lightning is secure, prior work has identified several attacks on the assumptions of Lightning and properties that are not included in our security definition. In particular, griefing is studied [Pér+20; MZ21a; RMT19; LHY22] as well as other denial-of-service attacks [TZS20] in which no funds are stolen but the normal operation of the payment channel network is disrupted. In the so-called wormhole attack [Mal+19; TMM20], an adversary reroutes a payment to capture the routing fees intended for intermediate nodes while leaving the transferred payment amount unaffected. Extending the TLA$^+$ formalization of Lightning to include routing fees would make it possible to model the

wormhole attack. However, because fees would also needed to be incorporated into the security property, this extension would substantially complicate the security property. For this reason, modeling routing fees was considered out of scope for this thesis.

Furthermore, several publications investigate attacks on user privacy [Her+19; vDam+20; RT20; Kap+21; Rom+21; KER21; BNT22; NT24] which is a property that is not included in the security definition employed in this work. Other works [HZ20; RN21; NKW21; SS23] examine attacks that violate the assumption that users can publish transactions on the blockchain in a timely manner. In practice, security vulnerabilities can also arise from discrepancies between implementations and the specification also based on implementations not following the specification, for example due to missing or incorrect verification checks [Rus19].

4.8.4. Evaluation of Protocol Modifications

Beyond proving that the formalization of Lightning fulfills the security property, the TLA$^+$ formalization can also serve to test proposed protocol modifications. In many cases, potential flaws can be identified efficiently by model checking only a restricted subset of the specification, for example, a single channel, and by verifying only lower-level invariants and temporal properties. Although such an approach cannot provide a full proof for a modified version of Lightning, it can accelerate protocol development by offering rapid feedback to developers.

To evaluate this capability, we deliberately introduced flaws into the Lightning formalization and confirmed that these flaws are detected by the model checker. As a concrete example, we examined whether so-called second-stage HTLC transactions could be eliminated by embedding their output conditions directly into the outputs of commitment transactions. Model checking revealed within a few minutes that this modification renders the protocol insecure, demonstrating that Lightning cannot be simplified in this manner.

4.8.5. Connecting the Specification to an Implementation

Multiple Lightning implementations have been developed and are actively deployed in practice. Although the TLA$^+$ specification of Lightning is not executable, it can nevertheless be used to validate the correctness of such implementations. Recently, Cirstea et al. [Cir+25] proposed a lightweight approach for connecting implementations written in imperative programming languages to a TLA$^+$ specification. Their approach collects traces of program executions from an implementation and uses the model checker TLC to check whether these traces conform to the behaviors allowed by the corresponding TLA$^+$ specification. Adapting their approach to Lightning represents a promising direction for future work.

4.9. Conclusion

In this chapter, we presented our approach to check that the TLA$^+$ specification of Lightning (see Chapter 3) fulfills the security property defined in Chapter 3. Because the specification's state space is too large for model checking, we defined a stepwise refinement that shows in four steps that the specification of Lightning is a refinement of the security property. For verifying the refinement steps, we used two main approaches: The first approach was an optimization of the model of time used in the specifications to reduce their state spaces. We defined this optimization approach for a general real-time specification in TLA$^+$ and proved the correctness of the approach. Therefore, the approach can be reused for other real-time specifications in TLA$^+$ beyond our use-case of Lightning. The second approach was a decomposition approach that allowed us to verify an abstraction between two specifications of a payment channel network by model checking specifications that model only a single payment channel of the payment channel network.

While we used proofs and model checking to verify the stepwise refinement, the most complicated steps were verified by model checking. On the one hand, model checking is a powerful verification approach because the refinements are automatically checked for all behaviors of concrete models. While it is hard to find flaws in hand-written proofs, model checking reveals all flaws that appear in behaviors of the specification for the models that are checked. While working on the content of this chapter, we learned from experience that after the point at which a specification and a refinement mapping have been defined that 'look correct' there is still substantial effort required to arrive at definitions that are actually correct. On the other hand, model checking is a limited verification approach because the specifications can only be checked for finite models. We could exhaustively explore the state space of models that defined payments that were sent over up to six hops and models with two subsequent or concurrent payments. Because we cannot generalize from our results that refinements hold for models with more than two payments, we used random simulation to partially check models with more than two payments and did not find any violations. However, a more complete verification is left for future work. Such a verification could employ other verification approaches such as theorem proving for unbounded verification.

In this and the previous chapter, we have presented an approach to verify that a payment channel network fulfills its security properties. We have modeled the Lightning protocol and we modeled an adversary that can crash and publish transactions on the blockchain that are not specified by the protocol. Our model checking results give strong confidence that, even in the presence of such an adversary, Lightning fulfills the security property that honest users finally receive their correct balance. Our approach can be adapted to analyze and verify other implementations of payment channel networks. Depending on how different such a different implementation of a payment channel network is, it might be that the higher-abstracted steps of the stepwise refinement can be reused and only the refinements on the lower-abstraction levels need to be adapted.

5. The Layer Below a Payment Channel Network

In this chapter, we focus on the layer below a payment channel network and we consider the research question whether the assumptions that payment channel networks make about their underlying layer are valid. To approach this question, we first define the RFL model (Reduced First Layer) that describes the assumptions that a payment channel network makes about its underlying layer. By reapplying the results and the methodology of Chapters 3 and 4, we find that the reduced model suffices as a first layer to securely implement a payment channel network. While payment channel networks are typically built on top of a blockchain, the properties of the RFL model are reduced compared to the properties provided by a blockchain. Specifically, while a blockchain makes each transaction visible for all other users, the reduced model restricts the visibility of transactions to the users who are affected[1] by a transaction. The RFL model shows that a payment channel network does not necessarily need a blockchain as first layer. We show that the RFL model could also be implemented using central parties like banks or payment service providers.

Previous work [GKL15; GKL20] has shown that Bitcoin fulfills the requirements that the Lightning Network has for its underlying layer. This is shown based on a set of assumptions on the peer-to-peer network used by Bitcoin. Empirically, it can be seen that these assumptions are met by the current peer-to-peer network of Bitcoin, however, little is known about the peer-to-peer network itself. Because of this knowledge gap, Bitcoin developers can hardly evaluate the effects of proposed changes and, thus, the development of code concerning Bitcoin's peer-to-peer network is risky and done only cautiously and slowly. In an empirical part of this thesis, we use and extend a peer-to-peer network monitoring infrastructure to collect and analyze evidence on previously unknown metrics about the peer-to-peer network such as the number of unreachable peers in the network (see Section 5.2.4) or the peer degree of reachable peers (see Section 5.2.5).

We conclude the chapter with an answer to the initial research question: While we define assumptions of payment channel networks about their underlying layer, we cannot completely show that Bitcoin fulfills these assumptions because the properties of Bitcoin rely on assumptions on Bitcoin's peer-to-peer network which are not always fulfilled. While we contribute to a better understanding of Bitcoin's peer-to-peer network, further analyzing

[1] In Section 5.1, we will define affected users as users who are able to spend either the transaction's outputs or the outputs referenced by the transaction's inputs.

and improving Bitcoin and Bitcoin's peer-to-peer network is a challenge for future work. In practice however, these theoretical results seem to be less relevant because, under usual circumstances, Bitcoin fulfills the assumptions of payment channel networks.

The content of this chapter is based on the following previous publications: The generalization of the first layer has previously been published in [GH20], the analysis of the peer degree distribution has been published in [GBH22], and the empirical analysis of unreachable peers has been published in [Gru+22]. In this chapter, we extend upon these results and put them into context.

5.1. Generalization of First Layer

In this section, we show that a payment channel network can be implemented on top of a first layer that offers weaker guarantees than a blockchain. To this end, we present the RFL model (Reduced First Layer) which is a model for a payment channel network's first layer with reduced properties compared to a blockchain. While a blockchain makes each transaction publicly visible to all users, in the RFL model, a transaction can be seen only by the users who are affected by the transaction. While properties of a blockchain have already been formalized by previous work [KZZ16; Bad+17], the novelty of our work is that we reduce the model so that it is weaker than a blockchain but still suffices to be used as a basis for a payment channel network.

We present the RFL model in Section 5.1.1. In Section 5.1.2, we adapt the TLA$^+$ formalization of Lightning and the model checking approach presented in Chapter 4. While we can only completely model check models with one payment, partial model checking of larger models indicates that the RFL model suffices as a first layer to implement a payment channel network that fulfills the security property defined in Chapter 3. We show in Section 5.1.3 that, under certain assumptions, Bitcoin implements the RFL model but also central parties like banks or payment service providers could implement the RFL model. Finally, in Section 5.1.4, we show that the properties of a blockchain in contrast to the reduced properties of the RFL model can be exploited to improve the properties of a payment channel network that is built on top of a blockchain. We conclude the section in Section 5.1.5.

5.1.1. Model for Reduced First Layer

In this subsection, we present the RFL model. The main difference of the RFL model to a blockchain is a modified visibility property: While in a blockchain each confirmed transaction is visible to all users, in the RFL model users see only those transactions that they are affected by. To define the RFL model, we also define the set of affected users of a transaction.

In the RFL model, there is a set of users U. Each user interacts with the first layer. In particular, each user can create transactions and observe transactions that are published on

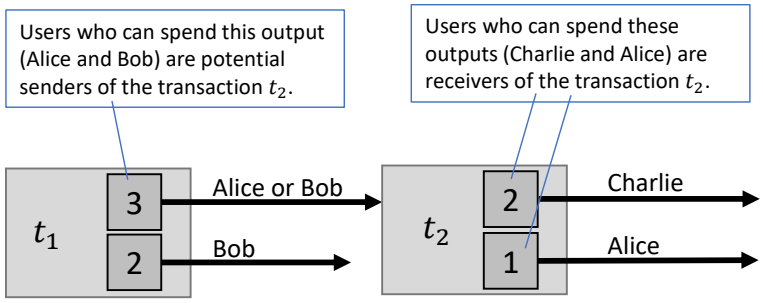

Figure 5.1.: Example of potential senders and receivers of a transaction of two transaction t_1 and t_2. The transaction t_2 spends the first output of the transaction t_1. The label at the arrow at each output indicates which users can spend the output.

the first layer. Transactions are modeled in the RFL model by an abstraction of the UTXO model (see Section 2.2). A transaction t is modeled as a record that contains a set of inputs and a set of outputs. Each output contains conditions that need to be fulfilled to spend the output. Such conditions can, for example, be that the spending transaction must be signed using a specific public key or that the preimage for a given hash must be presented. Each input refers to another transaction's output that is spent by the input and the input must provide the necessary information to fulfill the spent output's conditions.

We refer to the set of users who have the necessary secrets (e.g. private keys, preimages) to spend any output of transaction t as the *receivers* $\Omega_t^{\mathrm{R}} \subseteq U$ of transaction t (see Fig. 5.1). For this definition, we ignore timelocks of transactions, i.e., if a user can spend an output of transaction t at any future point in time, the user is a receiver of transaction t. Analogously, we refer to the set of users who have the necessary secrets to spend any output that is referenced by an input of transaction t as the *potential senders* $\Omega_t^{\mathrm{R}} \subseteq U$ of transaction t (see Fig. 5.1). The prefix 'potential' indicates that it can be that a user u is a potential sender of a transaction t although user u has not actually sent the transaction t but that transaction t is used to spend an output that user u could have spent. While users who created and signed a transaction t have seen the transaction t before it is published, there might be potential senders and receivers of the transaction t who do not know the transaction t at the time of the transaction's publication. We define the set of *affected users* Ω_t of transaction t as the union of potential senders and receivers of transaction t, i.e. $\Omega_t = \Omega_t^{\mathrm{S}} \cup \Omega_t^{\mathrm{R}}$.

We model the first layer $\mathcal{L}$ as a single logical party. Each user has a secure and reliable communication channel to the first layer $\mathcal{L}$. Users can interact with the first layer $\mathcal{L}$ by sending a *publish* message or a *query* message to the first layer: A user can *publish* a transaction by sending it to the first layer. A user will not get a reply from the first layer but a user can check whether the transaction was published by sending a *query* message to the first layer. If a user sends a *query* message to the first layer, the first layer responds with the set of published transactions. The responses of the first layer to these two requests need to fulfill the following properties: A transaction is only published if it is valid (*Transaction Validity*), the first layer will publish a transaction after a time interval

that is at most the confirmation delay Δ_{conf} (*Liveness*), published transactions will never be 'unpublished' (*Persistence*), and a transaction is only visible to the affected users of the transaction (*Affected User Visibility*). We define these properties as follows:

- *Transaction Validity:* If a user u sends a *publish* message with transaction t to the first layer $\mathcal{L}$ and the transaction t is not valid, the first layer ignores the transaction t. A transaction is valid if all inputs of the transaction contain the required data to fulfill the conditions of the referenced outputs and if the transaction does not spend the same output as a valid transaction that was sent to the first layer $\mathcal{L}$ before.
- *Liveness:* If a user u sends a valid transaction t to the first layer $\mathcal{L}$, then $\mathcal{L}$ will include the transaction t after a delay of at most Δ_{conf} in responses to subsequent queries by the same user u.
- *Persistence:* Each transaction t that is included in the response to a query for published transactions by user u is included in any response to a subsequent query by the same user u.
- *Affected User Visibility:* If the first layer $\mathcal{L}$ includes the transaction t with the set of affected users Ω_t in a response to a query of any user, then the first layer includes the transaction t in all subsequent queries for published transactions by any user $u \in \Omega_t$.

Note that the difference of the first layer to a blockchain such as Bitcoin is the affected user visibility property: A blockchain makes a transaction visible to all users. The first layer according to our definition must only make a transaction visible to the users who are affected by a transaction.

5.1.2. Security of Payment Channel Protocol on Top of Reduced First Layer

In this subsection, we show that the RFL model suffices to securely implement a payment channel network based on the results of the previous chapter. In Chapter 4, we have verified that the protocol of Lightning is secure using a model of a blockchain as first layer. We now show how to adapt the TLA$^+$ formalization of Lightning to use a model that conforms to the RFL model. By partially model checking the adapted TLA$^+$ formalization, we find support for the statement that Lightning is a secure implementation of a payment channel on top of the RFL model. This implies that the RFL model suffices to securely implement a payment channel network.

We first discuss to which extent the model used in the TLA$^+$ specification of Lightning fulfills the properties of the RFL model. The model of the first layer in the TLA$^+$ formalization of Chapter 4 already fulfills transaction validity and persistence. In the TLA$^+$ formalization, the operator *PublishTx* is used to add transactions to the ledger (variable *LedgerTx*). Transaction validity is modeled by the operator *PublishTx* because this operator checks that the transaction is valid before a transaction is added to the ledger. Persistence

holds because transactions are never removed from the variable *LedgerTx*. The model of the first layer of the TLA$^+$ formalization also fulfills liveness but only in a restricted form corresponding to $\Delta_{\text{conf}} = 0$, i.e. all published transactions are immediately confirmed. This liveness property is modeled by the operator *PublishTx* because this operator adds valid transactions immediately to the ledger. A second difference between the model used in the TLA$^+$ specification of Chapter 3 and the RFL model is that all users observe the entire ledger, whereas the RFL model restricts the visibility of a transaction to those users that are affected by the transaction.

To align the TLA$^+$ formalization with the requirements of the RFL model, we introduce two conceptual modifications. First, we extend the model of the first layer to enforce affected user visibility. We present how we implement this property in Section 5.1.2.1. Second, we model the confirmation delay Δ_{conf} during the publication of transactions. We present how we model the confirmation delay in Section 5.1.2.2. The TLA$^+$ formalization of Lightning presented in the previous chapter assumes that transactions become immediately visible. With the delayed visibility of transactions, this assumption does not hold anymore and two changes are required: The first required change is that the definition of time bounds in the module *PaymentChannelUser* needs to be modified. We describe these changes in Section 5.1.2.3. The second required change is that revocation transactions must spend only one output. This means that multiple revocation transactions must be created to punish a cheating transaction with multiple outputs. We describe these changes in Section 5.1.2.4. In Section 5.1.2.5, we present our approach for verifying the changed specification based on the model checking approach presented in the previous chapter. While we can only completely model check small models with a single payment, our successful results of partially model checking models with more payments support our claim that the RFL model suffices to implement Lightning.

5.1.2.1. Implementation of Affected User Visibility

We introduce an operator *AffectedLedgerTx(u)* that restricts the view of user u to see, instead of the set of all published transactions *LedgerTx*, only transactions t for which the user u is an affected user (formally: $u \in \Omega_t$). We replace all read accesses to *LedgerTx* in the module *PaymentChannelUser* of the Lightning specification with the result of *AffectedLedgerTx(u)*. This change ensures that a user sees only those transactions that affect the user. Because each Lightning transaction affects exactly the parties of the payment channel in which it occurs, each transaction is visible to both parties of the payment channel and, thus, this change does not alter the behavior of the protocol but ensures compatibility with the RFL model.

5.1.2.2. Implementation of Confirmation Delay

We model the confirmation delay Δ_{conf} as a constant numeric value. To model the effect of the confirmation latency on the first layer, we put a further wrapper around a user's access on the transactions in the ledger. The operator *AffectedLedgerTx(u)* already wraps

the variable *LedgerTx* and reveals only transactions t that are visible to a user u if the user is an affected user of the transaction t. We add a further operator *ConfirmedLedgerTx* that wraps the operator *AffectedLedgerTx(u)* and reveals only those transactions whose age is at least the confirmation delay Δ_{conf}. Thereby, we ensure that a user sees only those transactions on the ledger that are old enough so that they have been confirmed. This is a worst-case assumption because, by definition of the RFL model, a user might see a transaction already before the confirmation delay Δ_{conf} but the first layer guarantees only that transactions are visible after the confirmation delay Δ_{conf}.

5.1.2.3. Required Change in the Definition of Time Bounds

The addition of the confirmation delay requires subtle changes in the specification. The first of these changes is that the specification of time bounds in the module *PaymentChannelUser* becomes more complex. Consider the following scenario: In the TLA$^+$ specification of Lightning without a confirmation delay, each user who wants to publish a commitment transaction can see whether the commitment transaction can be published or whether the other user has already published a conflicting commitment transaction. In the adapted specification with a confirmation delay, a user who wants to publish a commitment transaction might not see a conflicting transaction on the first layer but might still not be able to publish the transaction because there is a conflicting transaction that has just been published and is not visible yet. To reflect such situations, the time bounds must include enough time for users to wait for transactions to become visible and then react to those transactions.

5.1.2.4. Required Change of Revocation Transactions

A further impact of the introduction of the confirmation delay is a race condition that can lead to a user having to wait multiple times for the confirmation delay Δ_{conf} until the user has completely punished a dishonest user. In the following, we describe such a situation: Imagine a situation with an honest user and a dishonest user in a payment channel. The dishonest user publishes an outdated commitment transaction that contains multiple HTLCs that have already timed out. Assume that these HTLCs are incoming HTLCs for the honest user and outgoing HTLCs for the dishonest user. Thus, the dishonest user has for each HTLC a timeout transaction that is signed by the honest user and can be published on the blockchain. Because the commitment transaction is outdated, the honest user can also spend all outputs using the corresponding revocation keys. Figure 5.2a illustrates this scenario by showing a simplified commitment transaction that contains four outputs for four HTLCs (other outputs are omitted). If the honest user observes the outdated commitment transaction on the blockchain, the honest user publishes a revocation transaction that spends all outputs of the commitment transaction. It might, however, be that the dishonest user concurrently publishes one of the HTLC timeout transactions that spends one output. If the HTLC timeout transaction is confirmed because it was published before the revocation transaction, the revocation transaction

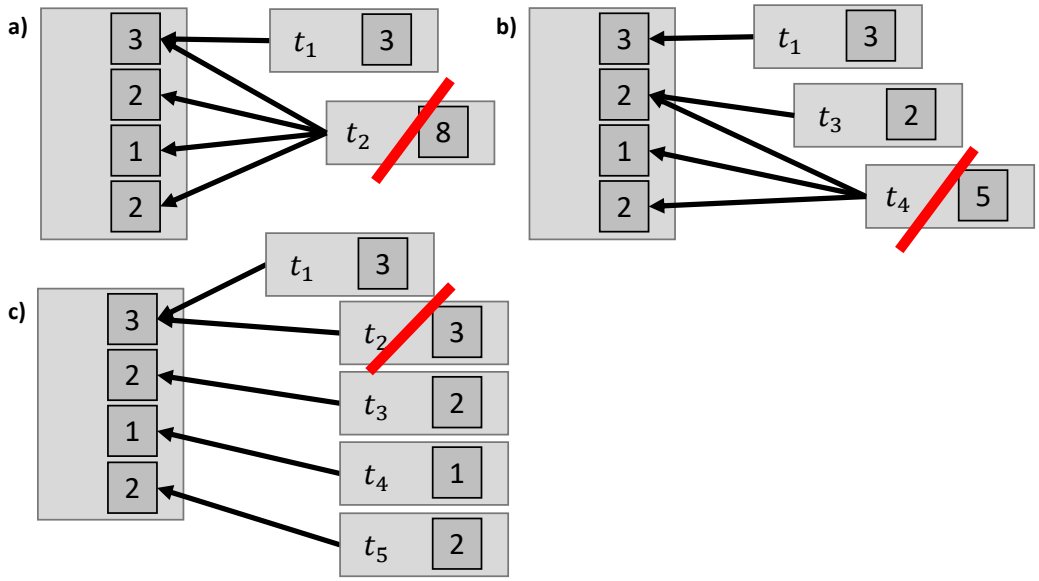

Figure 5.2.: Illustration of an issue with race conditions and our approach to prevent this issue. a) A commitment transaction might contain multiple outputs that are spendable by both parties of the channel. If one party publishes transaction t_1 spending the first output and the other party publishes transaction t_2 spending all four outputs and transaction t_1 is confirmed, the transaction t_2 will be invalid. b) The race of a) might continue with the remaining outputs: Transaction t_3 spends one output and transaction t_4 created by the other user spends the three unspent outputs. If transaction t_3 is confirmed, transaction t_4 will be invalid. c) The issue can be resolved by creating a dedicated transaction for each output that is spent. If two transactions (here transactions t_1 and t_2) spend the same output, all other outputs are nevertheless successfully spent.

becomes invalid because it spends an output that has already been spent by the HTLC timeout transaction (see Fig. 5.2a). After the confirmation delay Δ_{conf}, the honest user observes that the punishment attempt was not successful. However, the HTLC timeout transaction has an output that can be spent using the same revocation key as the associated commitment transaction and, thus, the honest user can still punish the dishonest user. If the honest user creates a second revocation transaction that spends all the remaining unspent HTLC outputs, the second revocation transaction might again conflict with an HTLC timeout transaction that was published shortly before the second revocation transaction (see Fig. 5.2b). Thereby, the dishonest user can delay punishment by the confirmation delay Δ_{conf} for each HTLC contained in the commitment transaction. This issue can be prevented if the honest user creates and publishes multiple revocation transactions: a single revocation transaction for each spendable output (see Fig. 5.2c). Thereby, if there is already a conflicting transaction in the first layer that was not yet visible for the honest user, all outputs for which no conflicting transactions exist are successfully spent. After waiting for the confirmation delay, the conflicting transactions become visible and can be handled. We adapt the TLA$^+$ specification of Lightning so that revocation transactions spend only a single output. The official specification of Lightning also warns about using

a single transaction for punishment[2]. However, before implementing the confirmation delay in the TLA$^+$ specification, this issue did not appear because in the model used by the specification each user had always a current view on the state of the blockchain.

5.1.2.5. Model Checking

We check that the TLA$^+$ specification of Lightning with these changes still fulfills the security property presented in Section 3.2. To this end, we use model checking. Because the changes affect only the lowest level of the stepwise refinement (i.e. *SpecificationI*), it suffices to show that the modified version of *SpecificationI* implements *SpecificationII*. To show that the modified version of *SpecificationI* implements *SpecificationII*, we use the same approach as in Chapter 4. Based on the time-optimization approach and the decomposition approach, we model check that $SpecificationI^{time,\ single}$ implements $SpecificationII^{single}$. Because of the change that transactions are only confirmed after the confirmation delay Δ_{conf}, there are more time points at which steps become possible and, thus, the time-optimized specification $SpecificationI^{time,\ single}$ has a substantially larger state space than in the case of Chapter 4. Therefore, the runtime for model checking is significantly higher and we can only exhaustively model check models in which a single payment is made. We use random simulation for partially model checking models with more payments and do not find counterexamples. While a more complete verification remains a challenge for future work, we conclude that the RFL model seems to be a sufficient abstraction of a first layer to implement a payment channel network that fulfills the security property we defined in Chapter 3.

5.1.2.6. Conclusion

By adding an operator to implement the affected user visibility and by adding a second operator to implement the confirmation delay, we adapted the TLA$^+$ specification of Lightning to use the RFL model. Because we could show by model checking that Lightning is secure also based on a first layer following the RFL model, we have shown that the RFL model suffices to securely implement a payment channel network.

5.1.3. Instantiating the RFL model

While the RFL model is an abstract model, we now discuss two instantiations of this model: First, we show that the Bitcoin blockchain implements the RFL model. Second, we discuss that the RFL model could be implemented by a more centralized first layer that is based on institutions like banks.

[2] `https://github.com/lightning/bolts/blob/master/05-onchain.md#revoked-transaction-close-handling`

5.1.3.1. Instantiation by Bitcoin

A design goal of the Bitcoin protocol is to fulfill guarantees that imply the properties of the RFL model. From a practical point of view, these properties seem to be fulfilled because, provided that users follow best practices[3], users can observe that these properties are fulfilled. However, this practical experience does not guarantee that Bitcoin fulfills the guarantees of the RFL model in the presence of adversaries with the goal to prevent Bitcoin from fulfilling the properties, e.g. an adversary might want to censor a transaction and thereby break the liveness property. A theoretical analysis by Garay et al. [GKL15; GKL20] shows that, under certain assumptions, Bitcoin fulfills the properties of the RFL model even in the presence of adversaries. In the following, we discuss these assumptions and find that, while they are not guaranteed to be fulfilled, in practice the assumptions are usually fulfilled and, thus, Bitcoin can be expected to fulfill the requirements of the RFL model.

Garay et al. show [GKL15; GKL20] that, assuming a bounded-delay network model and an honest majority of computing power, the Bitcoin protocol satisfies the properties *consistency* and *liveness*. The definition of Garay et al. of consistency [GKL20] implies the properties persistence and affected user visibility according to our definitions. Liveness, as defined by Garay et al. [GKL15; GKL20], assumes that all honest parties have a transaction and implies that the liveness and the transaction validity property of the RFL model are fulfilled. Bitcoin aims to achieve the assumption that all parties have a transaction by flooding new transactions in a peer-to-peer network so that all transactions are known by all peers. For proving liveness, Garay et al. assume that an unbounded number of transactions may be inserted in a block [GKL15, p. 307]. In the actual Bitcoin protocol, however, the number of transactions in a block is bounded by an upper bound for the size of a block. During times of high transaction load, transactions might not be immediately confirmed but need to wait until they are added to the blockchain. Such a waiting time is modeled in the liveness property of Garay et al. by a 'wait time' parameter [GKL15, Definition 5] and, in our liveness property, by the confirmation delay Δ_{conf}. Lightning depends on the assumption that transactions can be published within a certain time frame. It has been shown [HZ20; NKW21; Tsa+21] that payment channel networks can be attacked if an attacker can delay the publication of transactions on the blockchain further than the time frame required by Lightning. Therefore, the values of the 'wait time' and the confirmation delay Δ_{conf} should be chosen high enough so that times of congestion do not break the liveness property. The value of these parameters is high enough if it is longer than the duration of a possible congestion. Based on measurements of past confirmation times [Blo] a conservative choice for these parameters is 14 days. Further, the expected confirmation time of a transaction can be reduced by increasing the fees paid for the transaction.

[3] E.g, waiting for six blocks until a transaction is considered to be confirmed and paying fees according to the recommended fee level.

We conclude that, under certain assumptions, Bitcoin implements the RFL model. The required assumptions can be seen as fulfilled if users follow best practices and include security margins when choosing the parameters such as the confirmation delay.

5.1.3.2. Instantiation by Banks

The abstraction of the properties of the first layer from a blockchain enables exploring other design options for a first layer that is different from a blockchain. For example, if we assume that users put a certain amount of trust into institutions like banks, a network of banks can be used to implement the RFL model. To sketch how a first layer implemented by banks might be designed, we assume that banks offer the common features of banks but these parties might also be other institutions, e.g., payment service providers or other companies. From a bank customer's point of view, a bank offers an interface that implements the properties liveness (transactions are confirmed within a bounded time), transaction validity (only valid transactions are accepted), and persistence (confirmed transactions are not removed). The visibility of transactions usually matches our definition of affected user visibility: Users can see only incoming and outgoing transactions but not transactions of other users. In the context of banks, a transaction with multiple potential senders is, for example, a transaction that is sent from a joint account: Each user who has access to the joint account can send the transaction and all users with access to the joint account can see the transaction after it has been sent. Using such an interface of banks as a first layer, customers could build a payment channel network to perform transactions without communicating with their banks. While banks need to be trusted to correctly implement the RFL model, transactions in the payment channel network are hidden from the bank which improves the privacy of users similar to cash transactions. We assume that banks implement the UTXO model or an equivalent model to represent transactions.

A payment channel that is being opened between two customers of the same bank is opened by a transaction that sends funds to a special joint account between the two customers. Both customers can see the balance in this joint account and, each user who funded the channel, can see the transaction used to deposit funds into the channel. The funds in the joint account can only be spent by a transaction that is signed by both parties. Both parties hold commitment transactions that are signed by the other party and spend the funds of the joint account. The latest commitment transaction always corresponds to the current distribution of balances in the payment channel. Assuming that the bank implements a transaction model that allows for conditions using timelocks and hash functions, even HTLCs can be implemented inside a payment channel. If one of the users sends a commitment transaction to the bank, the balance in the joint account is redistributed to the two parties and the payment channel is closed. To disincentivize closing a channel with an outdated commitment transaction, a punishment mechanism can be implemented as in Lightning.

5.1.3.3. Discussion

We briefly discuss the two presented ways of instantiating the RFL model using either Bitcoin or banks. For the discussion, we compare the two ways with regard to several aspects:

Trust The trust required to believe that Bitcoin correctly implements the RFL model can be reduced to a set of assumptions based on a proof [GKL20]. These assumptions are, for example, that the largest share of computing power is contributed by honest miners and that the underlying network delivers messages with a bounded delay. As we have not designed a concrete protocol for a first layer of banks we cannot reduce the trust in banks to specific assumptions but users have to trust that banks correctly implement the RFL model.

Privacy While the use of payment channel networks itself can improve privacy by hiding transactions from the first layer, payment channel networks do not generally guarantee transaction privacy [Kap+21; RT20; Rom+21]. The privacy properties of a payment channel network are influenced by the privacy properties of the first layer because each payment channel is opened and closed on the first layer which might leak information. In comparison to a blockchain on which users are identified by pseudonyms, banks are forced by regulation to implement methods for customer identification. With regard to the bank or an institution that has access to the bank's data, transactions at a bank are not private. With regard the public, however, transactions at a bank are more private than on a blockchain because banks implement access control on their data and strictly restrict the visibility of transactions.

Liquidity To open a payment channel, users have to deposit funds into the payment channel. On a blockchain, users can only deposit funds that they have access to. On a first layer implemented by banks, however, banks could give users credit to open payment channels. Such an approach, similar to credit cards, could improve the liquidity in a payment channel network and, thus, the ability to route payments through the network.

Online requirement Payment channel protocols require users to observe the first layer and react within a limited time to cheating attempts. Using a blockchain as first layer, users need to be connected to the blockchain and analyze new blocks. Users who cannot fulfill this requirement can outsource this task to a watchtower (see Chapter 6). In a first layer implemented by banks, it could be implemented that a bank contacts a customer in case certain transactions are published because the bank knows each customer. Also, the bank could be a party that runs a watchtower service for its customers.

Currencies Blockchains such as Bitcoin implement their own decentralized currencies. Thus, a payment channel network built on top of Bitcoin can only be used to send payments in the Bitcoin currency. In a first layer of banks, centrally banked currencies could be made available for use in a payment channel network.

5.1.4. Advantages of Stronger First Layers

We have shown above that the RFL model suffices as a first layer for a payment channel network although it guarantees weaker properties than a blockchain with respect to the visibility of transactions. As payment channel networks such as the Lightning Network are built on top of the Bitcoin blockchain, one could ask whether the stronger visibility property offered by Bitcoin can be used to improve the protocol of a payment channel network. Interestingly, this question has an affirmative answer: If users do not only see transactions that they are affected by but also other transactions, the multi-hop payment protocol can be modified to reduce the duration during which funds are locked during a multi-hop payment.

The Sprites protocol [Mil+19] is a protocol for multi-hop payments that uses the same timelock for all hops on a route in contrast to Lightning in which the timelocks along a route are descending from the sender to the receiver. In Lightning, the duration that funds are locked for depends on the length of the route. In Sprites, however, this duration is constant. Because the global timelock in Sprites can be set as low as the lowest timelock in Lightning, the payment's funds are locked in Sprites for a shorter time which enables more efficient use of funds for routing payments. The mechanism used in Sprites is that the payment channels along a payment's route do not use an HTLC but instead refer to a global 'preimage manager': A preimage manager is a smart contract on the blockchain that simply keeps a list of entries that consist of a preimage and the time at which the preimage was listed. For making a multi-hop payment, in each payment channel a contract is added with the condition that a preimage is registered at the preimage manager before the timelock. Because all payment channels refer to the same preimage manager, the preimage manager synchronizes the outcome of these contracts in all payment channels and, thereby, it is ensured that a payment is either fulfilled in all hops along the route or in none of the hops. To use the global preimage manager, however, transactions adding a preimage to the preimage manager need to be visible for all participants and, therefore, the restricted visibility property of the RFL model is not sufficient for Sprites.

5.1.5. Conclusion

In this section, we have presented the RFL model that differs from the properties of a blockchain by restricting the visibility of transactions. We have used the results and the approach from the two previous chapters to show that the properties of the RFL model suffice to securely implement a payment channel network. The visibility of transactions is an aspect in which a blockchain like Bitcoin strongly differs from banks and payment

service providers who, in principle, make a transaction only visible to the transaction's senders and receivers but not to the general public. We discussed that the reduced visibility property makes the RFL model implementable for parties like banks and payment service providers. This could be a direction to implement a payment channel network for CBDCs (Central Bank Digital Currencies) which is an idea that is already being discussed [Ben+24; Ben+25]. With our work, we have shown that using payment channel networks based on alternative implementations of first layers is technically feasible. However, such an approach also comes with limitations such as the requirement that funds need to be locked in a payment channel and that, depending on the specific implementation of the payment channel network, the approach might have worse privacy properties than a direct use of the first layer. These questions indicate interesting directions for future research.

5.2. Empirical Measurements of the Bitcoin Peer-To-Peer Network

As discussed above in Section 5.1.3.1, a payment channel on top of Bitcoin relies on the assumption that Bitcoin fulfills the properties of the RFL model. These properties are transaction validity, liveness, persistence, and affected user visibility. From a practical perspective, observations show that during the last years Bitcoin fulfilled these properties given that the value for the confirmation delay Δ_{conf} is not set too strict. Specifically, transaction validity is fulfilled by Bitcoin because only valid transactions are included into blocks. Liveness is fulfilled because, given that a transaction pays enough fees, transactions are usually accepted within a few hours [Blo]. A risk for persistence is that transactions can be removed from the blockchain during reorganizations, i.e. if users discard blocks because there is an alternative chain that contains more work. Such reorganizations happen on average about twice a month, however, these temporary forks have typically a length of only one or two blocks. Because temporary forks are rare and are limited to a few blocks, accepting a block as confirmed if the block has six successors suffices to implement the persistence property of the RFL model. Bitcoin fulfills the affected user visibility property because all transactions are publicly visible.

However, instead of relying on observations for an assessment of Bitcoin, it is desirable to analytically show that Bitcoin fulfills these properties. Such an analysis has been conducted and resulted in a proof [GKL15; GKL20] that is again based on assumptions such as an honest majority of computing power and that each peer can broadcast a block to the other peers within a bounded delay. The assumption of an honest majority of computing power is difficult, if not impossible, to validate. In the following, we focus on the assumption with respect to the underlying network. The network used by Bitcoin is an overlay network that builds a peer-to-peer network on top of IP and overlay networks for private communication (namely Tor [DMS04], I2P [I2P], Cjdns [DeL26], Yggdrasil [Ygg]). We observe this overlay network by monitoring the network using a dedicated monitor node that connects to as many nodes as possible in the network. Our observations [NGH] show that the time it takes for blocks to be distributed in the network is mostly below a few seconds. However,

assessing the quality of the network more analytically is difficult because little about the network is known. For example, not only assumptions about the behavior of peers have to be made but even the size of the network and the topology of the network are unknown. We improve on the state of the art by measuring two previously unknown metrics about the Bitcoin Peer-to-Peer (P2P) network: The number of unreachable peers that participate in block propagation and the peer degree of reachable peers. While many questions about the Bitcoin P2P network are still left open, our measurements of these two metrics enable a more realistic design of models of the Bitcoin P2P network that can be used for analytical analyses.

We give background information on the Bitcoin P2P network in Section 5.2.1. We present previous measurements of the metrics that we measure in Section 5.2.2. We describe our measurement setup in Section 5.2.3. The approach for estimating the number of unreachable peers is presented and empirically evaluated in Section 5.2.4. The approach for estimating the peer degree of reachable peers and the number of reachable peers is presented in Section 5.2.5. We relate the estimate for the number of unreachable peers and the estimate of the peer degree of reachable peers in Section 5.2.6. We estimate the number of concurrently active unreachable peers based on the two measurements which leads to estimates that differ by only 13 % which indicates that the measurement results are consistent. We conclude in Section 5.2.7.

5.2.1. Fundamentals of the Bitcoin Peer-to-Peer Network

The Bitcoin peer-to-peer network consists of peers that are connected to each other. In this context, the term ‘peer’ refers to a running instance of an implementation of the Bitcoin protocol. In the peer-to-peer network, such peers are connected to other peers via the Internet; either directly via IPv4 or IPv6 or via an overlay network such as Tor. While there can be multiple peer-to-peer networks that match this definition, we refer in the following to the Bitcoin peer-to-peer network that is commonly described as the ‘Bitcoin mainnet’.

A peer has at least one address. Each address is either an IPv4 or an IPv6 address, or an address in one of the supported overlay networks. In the following, we consider only Tor as overlay network because the other overlay networks (I2P, Cjdns, Yggdrasil) are less used [Yeo; NGH]. A peer can have multiple addresses, e.g., if the peer is connected over multiple networks, and multiple peers can share an address, e.g., if the peers are behind the same NAT. In the following, we will for the most part make the simplifying assumption that each peer has exactly one address. We evaluate this assumption for reachable peers in Section 5.2.5.3 and find that estimating the number of reachable peers by counting their IP addresses overestimates the number of reachable peers by about 15 %.

To learn about the addresses of other peers, peers inform their neighbors about known addresses using ADDR messages. An ADDR message contains between one and 1000 entries of which each entry consists of an address, a port, a network identifier, a timestamp, and service flags. The service flags are a bitstring in which each bit indicates whether

a certain service is offered by a peer behind the respective address. While peers can request an ADDR message with addresses from a connected peer, peers can also send ADDR messages unsolicitedly. Peers running Bitcoin Core, the software that is run by the majority of peers [Yeo; NGH], announce one of their own addresses unsolicitedly once a connection is established and then on average once every 24 hours. We refer to these addresses as self announcements. The main purpose of solicited ADDR messages is that the peer who requested the addresses learns about other peers. Thus, peers simply store the addresses that are received in solicited ADDR messages. Self announcements, however, have the purpose of making a peer known in the network. Thus, an address received in an unsolicited ADDR message is not only targeted at the peer who receives the address but should be forwarded to the other peers in the network. To avoid forwarding irrelevant addresses, Bitcoin Core forwards self announcements only if they are routable, i.e., the address does not belong to an IP address range that is reserved for private use, and if the service flags indicate that the sending peer stores and offers upon request at least the most recent blocks. Bitcoin Core limits the sending of ADDR messages per connection to, on average, two messages per minute. Thus, self announcements that are received in multiple incoming ADDR messages can be batched in a single outgoing ADDR message. Because it cannot be distinguished from an ADDR message on its own whether the message was received solicitedly or unsolicitedly, Bitcoin Core uses the heuristic that the addresses in an incoming ADDR message are considered to be self announcements if the ADDR message contains ten or fewer addresses. In this section, we focus on such small ADDR messages that consist of ten or fewer entries.

During connection establishment, both peers exchange VERSION messages. A VERSION message contains, among others, the services offered by the sending peer and the peer's user agent.

The peers in the Bitcoin P2P network can be categorized into reachable peers and unreachable peers. While this categorization is subjective as a peer might be reachable for one set of peers and unreachable for another set of peers, we assume in the following that each peer falls into either category. By the default configuration of Bitcoin Core, each peer opens ten outgoing connections to reachable peers and a reachable peer accepts incoming connections until the peer has in total 125 connections.

5.2.2. Related Work

Several studies have been conducted already to measure metrics of the Bitcoin P2P network. In this section, we present studies that are related to the metrics measured by us, i.e. the number of unreachable peers and the peer degree of reachable peers.

The number of reachable peers has been measured [DPH14], [Par+19] and is continuously measured by Bitnodes [Yeo] and us [NGH] using the following approach of crawling the network: Starting with an initial set of known Bitcoin peers, the measurement node tries to connect to the known peers. From each reachable peer, the measurement node requests addresses of other peers. The measurement node tries to connect to each of these

addresses and, if an address can be used to reach a peer, adds this reachable address to the set of known peers and repeats the process by requesting addresses from this peer. While this approach is effective to measure the number of reachable peers, it cannot be used to measure the number of unreachable peers because, if an address cannot be connected to, it is unclear whether there is an unreachable peer behind the address or whether there is no peer with this address. In March 2026, there are about 10 000 reachable peers with IP addresses [NGH] and about 16 000 peers reachable over Tor [Yeo].

A study that attempted to estimate the number of unreachable peers was conducted in 2017 by Wang and Pustogarov [WP17]. They ran 102 reachable peers for seven days and observed the incoming connections. For each incoming connection, they checked whether the connecting peers was reachable. They observed on average about 10 000 unreachable addresses in a six-hour interval. Based on their observations, they estimate that there were at least 155 000 unreachable peers active in a six-hour interval. We will explain their estimation method below in Section 5.2.4.4.

In the absence of further academic studies, we also use a measurement of the Bitcoin developer Luke-Jr as a reference. Luke-Jr runs a website [Luk] that lists the number of unreachable and reachable peers. However, the methodology behind the website is not publicly documented. We compare our measurements to those by Luke-Jr in Section 5.2.4.4.

To the best of our knowledge, there is only a single study that considers the peer degree of reachable peers. In 2014, Miller et al. [Mil+15] exploited an information leak in the propagation of addresses to infer connections between reachable peers. They could infer the subgraph of reachable peers and calculated the peer degree distribution of this subgraph. They found that the majority of peers had a degree between eight and twelve. This result matches the expectation because, at that time, the default for peers was to open eight connections to other peers. The results that we present below are different to the results of [Mil+15] because our peer degree distribution includes not only connections between reachable peers but also connections between reachable and unreachable peers. While other approaches [Del+19; BKP14; NAH16; GNH19] have been proposed to infer parts of the topology of the Bitcoin P2P network, these approaches were too costly to be executed. Further, topology inference approaches for the P2P networks of other cryptocurrencies have been proposed [Cao+20; Wan+21; DRT19; Li+21]. These approaches were used to determine the peer degree distributions of the Bitcoin testnet [Del+19], Monero [Cao+20], and Ethereum [Wan+21].

5.2.3. Measurement Setup

We use a monitoring infrastructure for the Bitcoin peer-to-peer network. The monitoring infrastructure was setup in 2015 by Till Neudecker [Neu19] and we operate the infrastructure since 2019. Initially, two monitor nodes were set up at KIT that are continuously connected to all reachable peers in the Bitcoin peer-to-peer network. The monitor nodes receive and persist incoming messages. Besides sending PING messages to measure the

latency to other peers and requesting ADDR messages from their connected peers, the monitor nodes are passive and do not send messages to other peers after the initial handshake when a connection is established. The incoming messages received by the monitor nodes are announcements for blocks and transactions and ADDR messages that contain addresses of other peers in the network. Using the entries in the ADDR messages, the monitor nodes learn about peers in the network and try to open connections to peers that they are not connected to. The monitor nodes observe but do not actively participate in the propagation of transactions and blocks. Thus, the monitor nodes do not offer a useful service to the peers that they are connected to. Peers that would create incoming connections to the monitor nodes would expect to receive forwarded transactions and blocks. Because we do not want to impair the service quality of the Bitcoin peer-to-peer network, the monitor nodes do not accept incoming connections. However, the monitor nodes create outgoing connections because incoming connections are less expected to deliver useful services.

We operate two monitor nodes to be able to validate the approach by checking whether both monitor nodes make the same observations and to continuously monitor the network even when one monitor node is undergoing maintenance. Because both monitor nodes run in the network of KIT, we extended the monitoring infrastructure in 2019 with a third monitor node that is located in a different network (TU Berlin, later TU Darmstadt) to collect observations from a different vantage point.

For further collection of network data, we set up a few regular Bitcoin peers that participate in the propagation of transactions and blocks and accept incoming connections. As the monitor nodes, these peers also log their observations. While the monitor nodes have an unusual view of the peer-to-peer network because they are connected to many peers, we use these regular peers to get the view on the network of an exemplary regular peer.

5.2.4. Estimation of Number of Unreachable Peers

We use the setup described in the previous section to estimate the number of unreachable peers that participate in the propagation of blocks. This number of unreachable peers is relevant because unreachable peers influence the time for information propagation: Because unreachable peers connect to multiple reachable peers, they reduce the shortest path length between the connected peers to a maximum of two. Thus, a higher number of unreachable peers leads to a smaller average shortest path length which leads to faster information propagation [SZT22b]. Also, a continuous measurement of the number of unreachable peers is relevant for the security of Bitcoin because it might allow to detect attacks on Bitcoin that rely on introducing a noticeable number of unreachable peers.

Because unreachable peers are only directly observable for those peers that the unreachable peers connect to, the number of unreachable peers is inherently hard to estimate. The number of unreachable peers can be estimated by observing a fraction of unreachable peers and extrapolating the total number of unreachable peers as done by previous work [WP17] (see Section 5.2.2). This approach requires that assumptions have to be made on the behavior of unreachable peers such as the number of peers that they connect

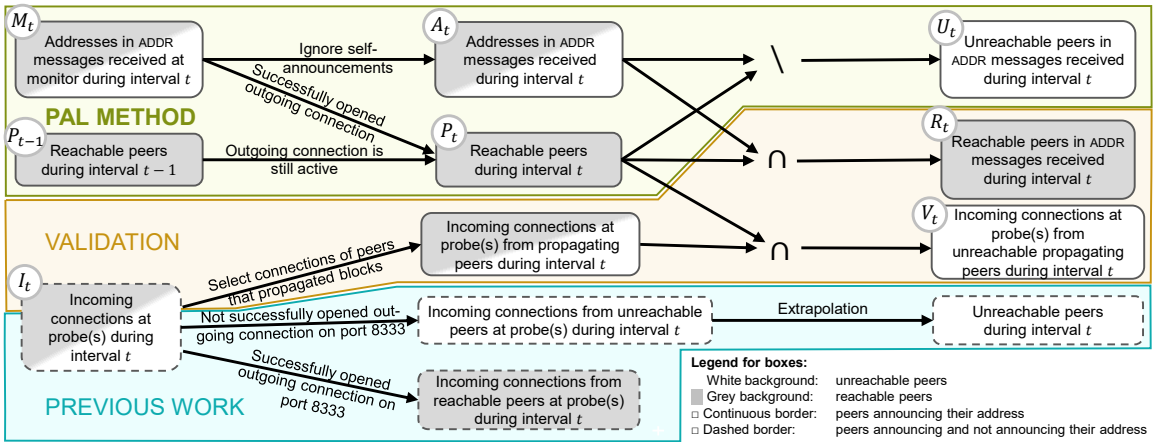

Figure 5.3.: Visualization of the PAL method, validation and the method of previous work [WP17]. In our measurements, we collect the sets M_t, I_t, and P_{t-1}. The arrows indicate filters and operations to derive more specific sets. A probe is a reachable peer that accepts incoming connections. Figure adapted from [Gru+22]. Reproduced with permission from Springer Nature.

to. In this work, we take a different approach of estimating the number of unreachable peers by observing effects caused by unreachable peers. In particular, we assume that unreachable peers that participate in the propagation of blocks announce their address to their peers. These announcements are propagated in the network and can be observed by our monitor peers. Based on these observations, we estimate the number of unreachable peers that are active per day. We refer to this approach as the PAL (Passive Announcement Listening) method. In the following, we describe this approach in more detail and validate the approach.

5.2.4.1. PAL Method

Data Collection We use a monitor node that is setup as described in Section 5.2.3. This monitor node connects to all known reachable peers and requests addresses from these peers. The monitor receives ADDR messages but does not propagate the received addresses. The monitor tries to connect to each received address to test whether the address is reachable. If a connection can be established, the connection is kept open until the connected peer closes the connection. The monitor logs received ADDR messages and VERSION messages as well as the points in time at which connections to other peers are opened or closed.

Data Analysis Based on the logs created by the monitor node, we use the following approach to estimate the number of unreachable peers. The approach is depicted in Fig. 5.3. We partition the logs into chunks so that each chunk corresponds to a time interval t. We refer to the set of all addresses that were received by the monitor node in the interval t as the set S_t. Starting from the complete set of received ADDR messages, we select only small ADDR messages, i.e. ADDR messages that contain ten or fewer entries. We refer to the

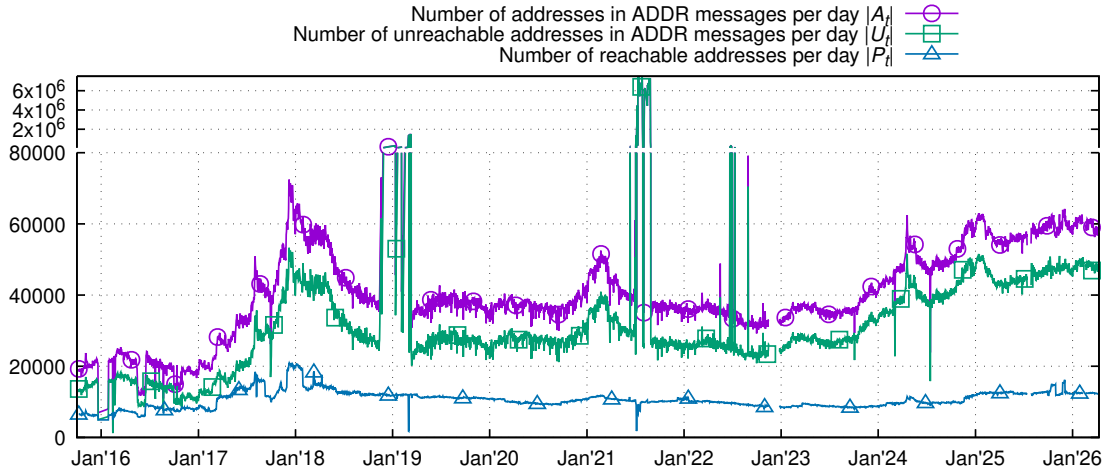

Figure 5.4.: Number of addresses observed in small ADDR messages compared to reachable addresses per day. Note that the upper part uses a different scale than the lower part.

resulting set as M_t (see Fig. 5.3). If the monitor is connected to an address a and receives an announcement of address a over this connection, we refer to this announcement as a self-announcement. We further reduce the set M_t by selecting only those addresses that are not self-announcements. The resulting set A_t contains only addresses received in small ADDR messages that were *forwarded* to the monitor node by one of the monitor node's connected peers. To select only addresses from *unreachable* peers in the set A_t, we use the following approach: We determine the set of reachable peers P_t in the interval t by determining the set of peers that the monitor was connected to at interval t. To this end, we combine the set of peers P_t that the monitor was connected to at the previous interval with the set of peers that a connection was opened to during the interval t and a VERSION message was received. We calculate the set $U_t = A_t \setminus P_t$ that contains only unreachable addresses. This set U_t is an estimate for the set of unreachable peers during interval t.

Limitations The PAL method measures the number of unreachable peers that announce their address while our goal is to estimate the number of unreachable peers that relay blocks. In Section 5.2.4.3, we validate how close the measured number is to the number of unreachable peers that relay blocks. A further limitation is that the measurement approach counts all peers that existed at least for a short moment during the interval t but cannot detect for how long a peer existed. Further, as addresses and the associated information are not authenticated, the received addresses might contain also bogus addresses that are not actually addresses of unreachable peers. We observed waves of such bogus addresses which we will discuss below.

5.2.4.2. Results

We implemented the described approach and used it to analyze data collected from 2015 to 2026 by one of the monitor nodes described in Section 5.2.3. We parameterize the time interval t with one day (we briefly discuss this choice below in Section 5.2.4.3). Figure 5.4 shows for each calendar day t the number of addresses received in small ADDR messages $|A_t|$ and the number of unreachable addresses $|U_t|$. The plot shows that the number of addresses in small ADDR messages $|A_t|$ varied over the years, ranging mostly between 20 000 and 60 000 addresses. Noticeable is a local maximum in December 2017 as well as several high peaks. An explanation for the local maximum in December 2017 might be that the price of Bitcoin had a local maximum at this time and many media reported on Bitcoin which might have resulted in many users temporarily running Bitcoin peers.

At the end of the year 2018 and early 2019, there is peak in the number of unreachable peers. Because such a sudden increase in the number of peers seems unlikely, we suspect that at this time addresses where distributed in the network that did not belong to actually running peers. To confirm this conjecture, we examined the addresses received in this time frame. However, we could not find any irregularities with regard to the addresses' distribution in the IPv4 address space, autonomous system, or country of autonomous system. Only for the highest peak in March 2019, we found that this peak was caused by many IP addresses that belonged to the same /8 subnet. Because IP addresses of this subnet were only rarely observed before, we assume that these IP addresses were caused by an unknown party flooding the network with these addresses.

In July 2021, an even higher peak of addresses could be observed. This peak was caused by addresses that did not belong to actual peers. Instead, we could observe that an unknown party sent bursts of IP addresses that were uniformly distributed over the IPv4 address space. We further discuss this peak in Section 5.2.5 where we use our observations of the propagation of these addresses to estimate the number and peer degree distribution of reachable peers.

During a few days in June and July 2022, a further spam wave was observed. This spam wave again contained IP addresses that were uniformly distributed over a large part of the IPv4 address space. We could not detect the same pattern as for the spam wave in 2021 and we did not conduct a deeper analysis of this spam wave.

A further local maximum can be seen in April 2024. The maximum coincides to the day with the date of the Bitcoin halving. At a Bitcoin halving, the reward that miners receive for mining new blocks is halved. Such a halving happens roughly every four years. Because around this date many media reported on Bitcoin, we attribute the increased size of the number of reachable and unreachable peers again to a temporarily increased interest into Bitcoin that lead to peers joining the network temporarily.

5.2.4.3. Empirical Evaluation

We evaluate the approach focusing on two main aspects: Firstly, we evaluate how well the approach actually measures the number of peers that announce their address. We find that the approach performs very well and counts about 95 % of unreachable peers that announce their address. Secondly, we empirically evaluate how the count of unreachable peers that announce their address is related to to the number of unreachable peers that participate in block propagation. In our evaluation, the PAL method detects propagating peers with a recall of 78.5 % and a precision of 90.6 %.

Announcing Peers Evaluating whether the monitor node receives an address of all peers that announce their address on that day is difficult because we do not know the ground-truth that we could compare our results to. Therefore, we evaluate the approach in two settings for which we know the ground-truth: Firstly, we run our own unreachable peer continuously for multiple months and evaluate whether the peer's address is observed by the monitor at every day. We find that the address of the unreachable peer was observed by the monitor on 200 of 212 days between December 2020 and June 2021 which corresponds to a probability of 94 % of observing the peer's address on each day. After the unreachable peer was shut down, the monitor did not observe the unreachable peer's address anymore, i.e., we found no false positives. Secondly, we apply the method to reachable peers and evaluate whether on each day the addresses of all reachable peers are observed by the monitor excluding observations of peers announcing their own address. In the context of Fig. 5.3, this means that we compare the set R_t to the set P_t. If the PAL method worked perfectly, both sets should be equal. We compare these sets for all days of the year 2020 and find that on average 95.4 % of the addresses of peers that were reachable on a day were observed in small ADDR messages on the same day (excluding self-announcements).

Further, we analyzed the effect of our choice of setting the interval length to one day and found that increasing the length of the interval t from one day to five days increases the share of observed reachable peers from 95.4 % to 96.1 %. However, with an interval length of one hour only on average 84.9 % of the addresses of reachable peers were received in an ADDR message in the same hour. This indicates that the interval length of one day is a reasonable trade-off.

Unreachable Block Propagating Peers To evaluate whether the PAL method gives a good estimate for the number of unreachable peers that participate in block propagation, we compare the results of the PAL method to a known ground-truth. We obtain a ground-truth of unreachable peers that participate in block propagation by running a reachable peer and observing incoming connections (see [WP17]). Bitcoin Core, the reference implementation of Bitcoin, opens not just persistent connections but also short-lived connections with the purpose of testing whether the other peer is reachable. As during these short-lived connections even peers that, in general, propagate blocks do not announce blocks, we consider only incoming connections that lasted at least one minute. From these incoming connections, we select those connections over which at least one announcement for a block

was received. Then, we remove connections from the set that are from reachable peers (see Fig. 5.3). The resulting set (V_t in Fig. 5.3) is a set of unreachable peers that connected to our reachable peers and propagated blocks. This set V_t serves as our ground-truth. This ground-truth has a limited scope because the set V_t does not contain all unreachable peers that propagate blocks but only those that connected to our reachable peer. To make our results comparable, we reduce the set U_t of addresses of unreachable peers received in ADDR messages to the same scope. We calculate $U_t \cap I_t$, i.e. the intersection between the unreachable peers observed in ADDR messages and peers that were observed by our reachable peer. Then, we compare our measurement set $U_t \cap I_t$ to the ground-truth set V_t for each day of the year 2020. We find that the PAL method had on average a recall of 78.5 % and a precision of 90.6 %. This means that the probability for the PAL method for detecting an unreachable peer that propagated blocks was on average 78.5 % and that on average 90.6 % of the peers that were detected by the PAL method actually did propagate blocks. The main reason for the false negatives that reduce the recall is that about 15 % of unreachable peers that propagate blocks do not announce their address and are, thus, not detectable by the PAL method. A further reason is that, even if unreachable peers announce their address, the probability to detect these peers is only about 94 % as shown above. 10 % of the peers detected by the PAL method are false positives. We spot-checked the false positives and found that many had non-standard user agents which indicates that these peers behave differently than we expect for a regular Bitcoin peer.

For the analysis above, we used the data from one of our two monitor nodes that run in the network of KIT in Karlsruhe, Germany. For further validation from another vantage point, we are running a third monitor node since 2019. The third monitor node replicates the setup of the first two monitor nodes that is described above but runs in a different physical location and a different autonomous system. We expect that the addresses received by the monitor nodes running at different locations should largely overlap if the measurement method is reproducible. By analyzing the addresses received by a monitor node running in Karlsruhe and by a monitor node running in Berlin between 2019 and 2022, we found that 96 % of the addresses received per day overlap. We conclude that the measurement is reproducible and that the observations of one monitor node are representative.

We conclude that the PAL method reliably detects unreachable peers that announce their address. However, based on the precision and recall determined in the evaluation, we estimate the number of unreachable peers that propagate blocks to be $\frac{90.6\,\%}{78.5\,\%} - 1 = 15.4\,\%$ higher than the number measured by the PAL method.

5.2.4.4. Comparison to Previous Measurements

There is no publicly known ground truth that we could compare our estimation to. Instead, we compare our estimation to previous measurements and estimations. In 2016, Neudecker et al. [NAH16] simulated the Bitcoin P2P network and, based on the simulated propagation behavior, they estimated that the P2P network had about 16 000 unreachable peers participating in transaction propagation. For the year 2016, the size of the set A_t is on average about 14 000. Based on the evaluation results, we estimate the number of unreachable

peers to be $14\,000 \times 1.154 \approxeq 16\,150$. This estimate and the estimate by Neudecker et al. fit well together. Because Neudecker et al. estimate the size of a network at a given point in time but we estimate the number of unreachable peers during a whole day, it is plausible that our estimate is slightly higher.

In May 2017, Wang and Pustogarov [WP17] estimated the number of unreachable peers (see Section 5.2.2). Their estimation approach was to run 102 reachable peers for a week, to observe incoming connections at these reachable peers, and to extrapolate the number of unreachable peers based on their observations (see PREVIOUS WORK in Fig. 5.3). Wang and Pustogarov estimated at least 155 000 unreachable peers to be active in each six-hour interval. This estimate is significantly higher than our estimate. To discuss this difference, we first explain the estimation approach of Wang and Pustogarov. From the short description in [WP17], our interpretation is that they used the following approach: They split the measurement period of a week into periods of a duration of six hours. They assumed that an unreachable peer makes, in each six hour period, on average 3.5 outgoing connections to reachable peers that are uniformly chosen from the set of all reachable peers. Let the number of unreachable peers be n. The total number of connections of unreachable peers is $3.5n$. Assuming a total number of 5540 reachable peers[4], the average number of connections from unreachable peers seen at a single reachable peer in each six hour period is $\frac{3.5n}{5540}$ because of the assumption that unreachable peers choose the peers to connect to uniformly. Wang and Pustogarov ran 102 reachable peers and observed in each six hour window incoming connections from on average 10 000 unreachable addresses at all of their reachable peers, i.e. on average each of their peers observed $\frac{10\,000}{102}$ addresses of unreachable peers. Solving the equation

$$\frac{3.5n}{5540} = \frac{10\,000}{102}$$

for the number of unreachable peers n leads to

$$n = 10\,000 \times 5540 \times \frac{1}{102} \times \frac{1}{3.5} = 155\,182$$

which corresponds to the estimate given in the paper [WP17]. Because the measurement period is only a week and Bitcoin peers keep their outgoing connections as stable as possible, this estimation approach can only detect unreachable peers that joined the network within that week or peers that opened a new outgoing connection because an existing outgoing connection has been closed. Thus, the actual number of unreachable peers at that time would be expected to be even higher. However, the estimated value is already much larger than our estimate for the number of unreachable peers: At the time during which Wang and Pustogarov ran their experiment, we observed on average 29 200 unreachable addresses. This observation leads to an estimate of $29\,200 \times 1.154 \approxeq 33\,700$ which is less than a quarter of the estimate by Wang and Pustogarov. A possible explanation for this difference is the type of peers observed by Wang and Pustogarov. Wang and Pustogarov report that the 93.9 % of their connections where short-lived connections

[4] The origin of this number is unclear. It might have been obtained from Bitnodes [Yeo].

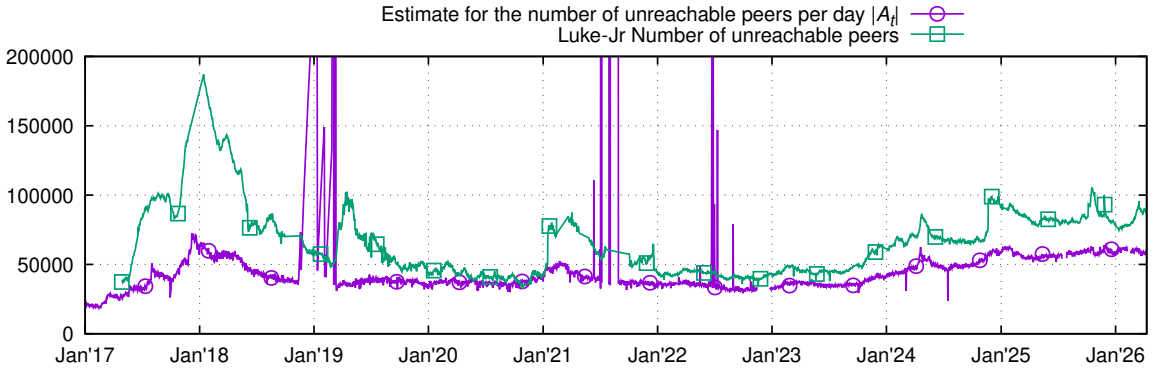

Figure 5.5.: Estimate for the number of unreachable peers using our approach compared with the estimate for the number of unreachable peers by Bitcoin developer Luke-Jr [Luk]. Our estimate is $|A_t|$ for each calendar day t.

that lasted less than 60 seconds and that about 80 % of peers could be classified by their user agents (BitcoinWallet, Breadwallet, and Bitcore) as mobile peers. We inspected the source code of these Bitcoin implementations and found that these implementations do not announce a peer's address. Thus, the mobile peers were invisible to the PAL method. These mobile peers are also less relevant for the peer-to-peer network because they do not participate in transaction propagation to conserve bandwidth.

Bitcoin developer Luke-Jr regularly publishes his own estimations of the number of unreachable peers [Luk]. Because of the lack of other points of comparison, we compare our results to the estimations of Luke-Jr although the methodology for his estimation is not publicly documented. Figure 5.5 shows the results of the PAL method and the estimates of Luke-Jr. While, for the most part, the estimates of Luke-Jr are higher than the estimate based on the PAL method, the shapes of the graphs for both estimates show similarities. On average per calendar day, the estimate of Luke-Jr is 47 % higher than the estimate of the PAL method. Based on the fact that the estimates of Luke-Jr do not contain the high peaks that are visible in our measurements, we suppose that Luke-Jr uses an approach that is not solely based on observing addresses. Therefore, we interpret the comparison as confirming that the measurements obtained with the PAL method reflect both the trend and the approximate number of unreachable peers.

5.2.4.5. Conclusion

We presented the PAL method as an approach to estimate the number of unreachable peers based on listening for ADDR messages that contain self announcements. Our measurements show that between 2015 and 2026 the number of unreachable peers was mostly between 20 000 and 50 000. Our empirical evaluations showed that the measurement approach has a precision of about 95 % to detect peers that announce their address. Further, we found that the set of unreachable peers obtained by the PAL method can be interpreted as an estimate for the set of unreachable peers that propagate blocks with a recall of 78.5 % and a precision of 90.6 %.

If the Bitcoin P2P network is modeled, for example, for simulating information propagation (see [NAH16]), our results can inform the choice of parameters. As we discussed when we compared the PAL method to previous work, the term 'number of unreachable peers' can be instantiated with different meanings. While the PAL method measured the number of unreachable peers per day that announce their address, the number of unreachable peers that are part of the P2P network at a specific moment can be different.

5.2.5. Estimation of Peer Degree Distribution and Number of Reachable Peers

In 2021, we observed a spam wave in the Bitcoin P2P network during which a high number of addresses were propagated (see Section 5.2.4.2). The spam originated from a set of spamming peers. These spamming peers connected to all reachable peers and sent bursts of 500 ADDR messages within a short time. We will discuss more of our observations in Section 5.2.5.1. The reachable peers that received the spam addresses forwarded them to their peers. Based on our observations of forwarded addresses, we inferred the peer degree of publicly reachable peers and found that many reachable peers have a peer degree around 125. The estimation will be presented in more detail in Section 5.2.5.2. Because most peers run Bitcoin Core and 125 is the default maximum number of connections in Bitcoin Core, this indicates that the connection slots of many reachable peers are fully occupied. Further, the observations allowed us to group addresses that belong to the same peer which we will elaborate on in Section 5.2.5.3. Thereby, we found that simply counting addresses to estimate the number of reachable peers overestimates the number of reachable peers by 15 %. To validate these results, we compare the results to a ground truth for reachable peers operated by us where we know each peer's peer degree as well as the peer's number of addresses and find that the results are quite accurate.

5.2.5.1. Observations and Interpretation

Observations We observed the spam wave in July and August 2021 at our reachable peers. During this time, about 400 times a spamming peer connected to one of our reachable peers, sent 500 ADDR messages containing ten addresses each, and disconnected. We observed 243 different IP addresses of such spamming peers that conducted this spamming. All 5000 addresses that were sent in such a batch had the same timestamp which had a value that was up to nine minutes in the future. The addresses sent by the spamming peers were IPv4 addresses that were uniformly distributed over the IPv4 address space.

Interpretation The fact that the addresses sent by the spamming peers were uniformly distributed over the IPv4 address space indicates that the addresses were randomly chosen instead of being addresses of actual peers. Our interpretation of this behavior is that the spamming entity intended the addresses to be forwarded in the P2P network as long as possible. This interpretation is based on the following two arguments: First, the spamming

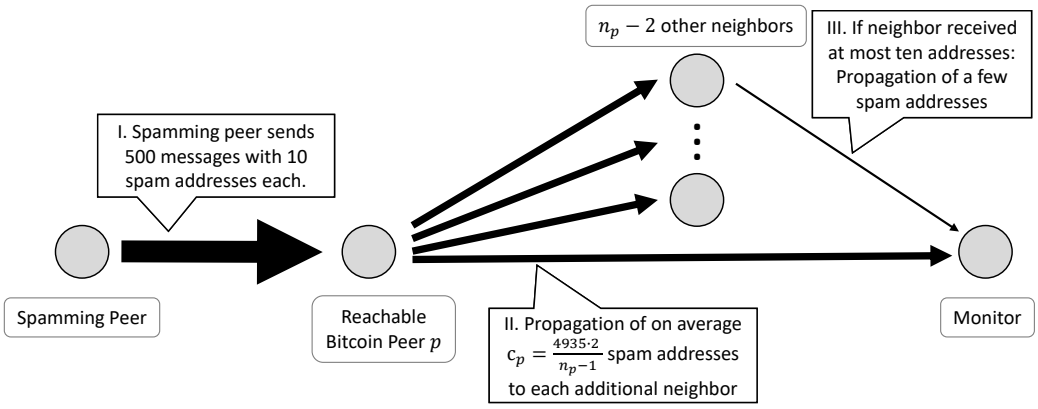

Figure 5.6.: Approach for estimating the peer degrees. A spamming peer connects to a reachable peer p and sends 5000 spam addresses (I). The reachable peer p is connected to our monitor node and to other nodes and has in total n_p connections. The peer p forwards each routable address to two of these nodes (II). From the number of addresses received at the monitor, we can estimate the peer degree n_p of peer p. Figure first published in [GBH22].

peers sent 5000 addresses within 500 ADDR messages although 5 ADDR messages would be sufficient to sent 5000 addresses because an ADDR message can contain up to 1000 addresses. Bitcoin Core forwards only those addresses that are received in an ADDR message that contains at most ten entries. Thus, by distributing the 5000 addresses over 500 messages it was achieved that the addresses were forwarded. Second, Bitcoin Core forwards addresses only as long as their timestamp is less than ten minutes old. Thus, typically an address is propagated for ten minutes. However, Bitcoin Core accepts addresses with a timestamp that is up to ten minutes in the future. Thus, by setting the timestamp to nine minutes in the future, the spamming entity achieved that the addresses were propagated for up to 19 minutes. Thus, it seems that the spamming entity intended that the addresses were propagated in the network for as long as possible.

5.2.5.2. Estimation of Peer Degrees

A reachable peer who receives a spam address chooses, equally distributed, two of its neighbors to forward the address to. As our monitor node connects to all reachable peers (see Section 5.2.3), the spam addresses are also forwarded to our monitor node. The main idea for the peer degree estimation is that the number of addresses received by our monitor from a peer correlates with this peer's degree. This estimation approach has already been described in 2014 [BKP14, Section 10.1], however, to the best of our knowledge, the approach has not been applied to the Bitcoin P2P network before. In the following, we explain this approach in more detail and discuss the results.

Figure 5.6 gives an overview of the setup and mechanisms behind the degree estimation. A spamming peer connects to a reachable peer p which has a total of n_p neighbors. One of these neighbors is our monitor node, another neighbor is the spamming peer, and there are $n_p - 2$ other neighbors. The spamming peer sends 500 ADDR messages

that contain ten addresses (see Fig. 5.6, I). Most reachable peers run Bitcoin Core, the reference implementation of Bitcoin [NGH]. To simplify the explanation, we assume in the following that all reachable peers run Bitcoin Core. Bitcoin Core forwards addresses that are received in ADDR messages with ten or fewer entries. However, addresses are only forwarded if they are considered routable, i.e. if they are not from IP address ranges that are reserved for private use. About 1.3 % of the IPv4 address space are considered unroutable, e.g., the subnet 10.0.0.0/8. Because the 5000 received addresses are distributed uniformly over the whole IPv4 address space, a reachable peer p forwards on average only $5000 \times (1 - 0.013) = 4935$ addresses. Each address is forwarded to two neighbors out of the $n_p - 1$ neighbors that are different from the peer that the address was received from. Thus, each neighbor receives on average $c_p = 4935 \times \frac{2}{n_p - 1}$ addresses (see Fig. 5.6, II). From the c_p addresses received at our monitor, we can estimate the peer degree n_p.

Because all reachable peers receive multiple batches of 5000 addresses, we make multiple observations of c_p for each peer p at our monitor. As all addresses in a batch have the same timestamp t, we use the timestamp t to distinguish the spam batches. We refer to the number of addresses received from peer p with the timestamp t as $c_{p,t}$. Our monitor nodes receive not only addresses that are directly forwarded after a reachable peer has received them from a spamming peer, but also addresses that are indirectly forwarded by other reachable peers (see Fig. 5.6, III). To filter out these indirectly received messages, (1) we consider only ADDR messages in our analysis that contain at least four entries, (2) we consider only addresses that have a timestamp that is between three and ten minutes in the future from the point in time when the ADDR message was received by our monitor, and (3) we consider only addresses received from peer p with timestamp t if $c_{p,t}$, the number of addresses received from peer p with timestamp t, is greater than ten. From each observed value $c_{p,t}$, we calculate $n_{p,t} = 1 + 4935 \times \frac{2}{c_{p,t}}$. We obtain the estimate for the degree of peer p by calculating the median of all $n_{p,t}$ for peer p based on addresses that were observed at the same day. The duration of a day is chosen as a trade-off between a short time during which a peer's degree remains constant and a long time window during which more observations are collected that make the estimate more precise.

We validate the approach with three of our own reachable peers that received addresses from the spamming peers and for which we logged the peers' degrees. We take each peer's average connection count during a day as ground truth. For each day during which the spamming continued, we compute the mean deviation of the estimate n_p to the ground truth in percent and average the absolute percentage points. By this calculation, we get an average deviation of 4.1 % of the estimate from the ground truth which means that the estimation is reasonable reliable.

By applying the method for each reachable peer in the network, we get for each reachable peer one estimate of the peer's degree per day. From all estimates obtained during the observation time period, we calculate a histogram that shows the estimated peer degree distribution in the Bitcoin P2P network (see Fig. 5.7). The histogram shows that about half the reachable peers in the network have a peer degree around 125. Assuming that they are running Bitcoin Core in the default configuration this means that all of their connection

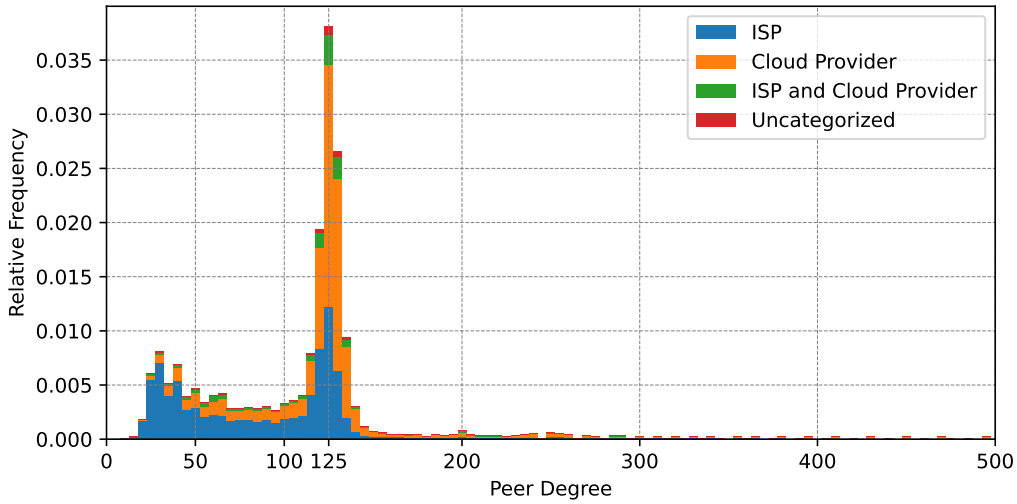

Figure 5.7.: Normalized histogram with bin width of 5 of the estimated peer degrees of all reachable peers based on our observations of the spam wave observed in July and August 2021. The peak at 125 represents $5 \times 3.8\,\% = 19\,\%$ of all peers. The different colors indicate our categorization of a peer's autonomous system which shows that the peer degrees of peers running at cloud providers tend to be higher than the peer degrees of peers running in the networks of ISPs. First published in [GBH22].

slots are filled. The histogram also shows that there is a long tail of a few peers with a higher degree than 140.

We determined the autonomous system (AS) associated with each peer's address using Team Cymru's IP to ASN Mapping Service [Cym]. We classified each AS into one of four categories: 'ISP' (Internet Service Provider), 'Cloud Provider', 'Both', or 'Uncategorized'. For ASes hosting a substantial number of peers, we performed a manual classification; for the remaining ASes, we obtained the category from the ASdb [Ziv+21] database. The distribution of peer degrees grouped by AS category is shown in Fig. 5.7. The results show that peers hosted at cloud providers tend to have a higher peer degree than peers located in networks by ISPs. Specifically, the median of estimated degrees for peers hosted at cloud providers is 125 whereas the median of estimated degrees for ISP-hosted peers is 97. A plausible explanation for this is that peers running in data centers accumulate more incoming connections because they are less often restarted. Additionally, their IP addresses are likely less often changed resulting in a broader dissemination of the address within the P2P network.

Validation Our results for the peer degrees show that a large share of peers has a peer degree around 125 and we suppose that these peers have all of their available slots for incoming connections filled. To validate this observation, we conduct an experiment to test the availability of slots for incoming connections at reachable peers. A reachable peer running Bitcoin Core always accepts an incoming connection. However, if the incoming connection occupies the last available slot, one of the peer's connections is evicted. The evicted connection might be the newly established one or a previously existing connection.

In our experiment, we deploy a test peer that iteratively attempts to connect to all reachable peers. For each reachable peer, the test peer establishes a TCP connection and waits three seconds before verifying whether the connection remains open. If the connection persists, the test peer initiates four additional TCP connections to the same reachable peer, waits another three seconds, and then checks whether all five connections are still open.

We conducted the experiment in November 2021 using three test peers located in two distinct ASes. To compile the list of reachable peers, we collected all addresses received in unsolicited ADDR messages at one of our monitor nodes on the preceding day. On average, our test peers successfully connected to 9461 peers. Of these, 4493 peers (47 %) accepted all five connections, 2360 peers (25 %) accepted the first connection but not all five connections, and 2608 peers (28 %) evicted already the first connection. These results indicate that 28 % of reachable peers have all incoming connection slots occupied, while 25 % of the reachable peers are near capacity. Only 47 % of reachable peers appear to accept five incoming connections without restriction. This finding supports our interpretation of the peer degree distribution and demonstrates that incoming connection slots constitute an actually limited resource within the Bitcoin P2P network.

5.2.5.3. Estimation of Number of Reachable Peers

When estimating the number of peers in the Bitcoin P2P network, a common assumption is that each IP address corresponds to exactly one peer (see [DPH14; FOA16; DBG18; Pap+18; Neu19; Par+19]). Above, we have also made this assumption when we estimated the number of unreachable peers based on the number of propagated addresses. However, this assumption does not hold in practice. There is no general, practical method to reliably identify multiple IP addresses that belong to the same peer. Our observations of the spam wave provided an opportunity to obtain a grouping of IP addresses to peers which led to the finding that counting reachable IP addresses overestimates the number of reachable peers by about 15 %.

Idea According to our observations, each batch of 5000 addresses that was sent by the spamming peers had the same timestamp. Each combination of 5000 addresses and timestamp was sent to a single peer, i.e. there were no two different peers that received the same 5000 spam addresses with the same timestamp. Thus, a batch of spam addresses with the same timestamp can serve as a unique marker for a specific peer. We can use these markers to find different IP addresses that belong to the same peer (see Fig. 5.8).

Method If a reachable peer has multiple IP addresses, the peer forwards the spam addresses across all its IP interfaces (Fig. 5.8, II). If the monitor receives identical tuples of spam address and timestamp from two different IP addresses, we infer that these addresses belong to the same peer (Fig. 5.8, III). If spam addresses propagate through multiple hops (Fig. 5.8, IV), false positives may occur. To mitigate this, we apply two filters: (1) We exclude spam addresses that are received in ADDR messages containing ten or fewer addresses.

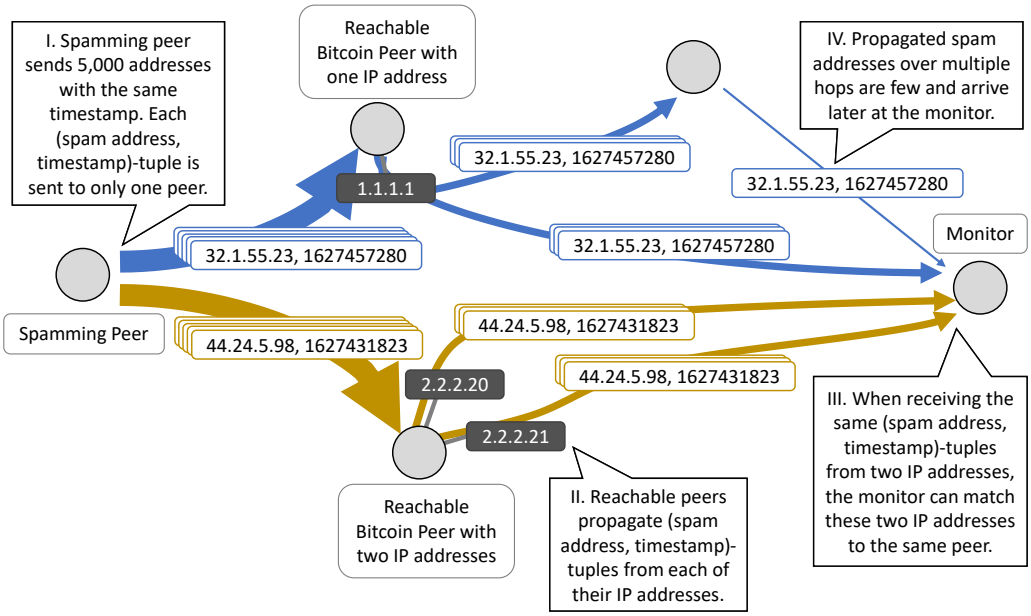

Figure 5.8.: Approach to match IP addresses to the same reachable peer. A peer with multiple reachable IP addresses is connected to the monitor through each of these addresses. If the monitor repeatedly receives the same tuple of spam address and timestamp from two different IP addresses, we assume that these two IP addresses are associated with the same peer. Figure adapted from [GBH22].

(2) We exclude spam addresses that have a timestamp that is less than five minutes in the future. Furthermore, we only match two IP addresses to the same peer if more than five identical (spam address, timestamp)-tuples are observed from both addresses.

Results and Validation We apply this method to data collected by our monitor nodes and identify sets of IP addresses that belong to the same peers. Each of these sets contains two IP addresses. If an IP address is included in multiple sets, we merge these sets. Thereby, we obtain a mapping of 3614 IP addresses to 1449 peers. While one peer appears to use 286 IPv6 addresses within the same /118 subnet, the majority (89 %) of peers have only two addresses. Most pairs consist of one IPv4 and one IPv6 address, although some pairs include two IPv4 or two IPv6 addresses. We validate the method using three of our own peers and find that their IPv4 and IPv6 addresses were correctly matched.

Impact In August 2021, our monitor nodes were connected to an average of 8647 reachable IP addresses per day. Of these, approximately 2220 IP addresses belonged to 1091 of the 1449 peers with multiple IP addresses. Consequently, the average number of unique reachable peers per day was 7518. This demonstrates that estimating the number of reachable peers by counting reachable IP addresses overestimates the number of peers by $\frac{8647-7518}{7518} \approx 15\,\%$

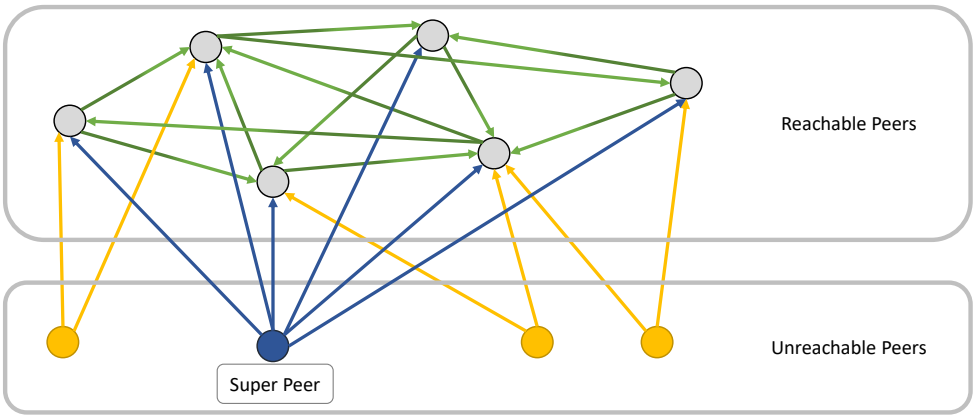

Figure 5.9.: Example of the model of the P2P network that we use to validate our results. In the simplified example shown in this figure, each reachable peer has two outgoing connections. Super peers are connected to all reachable peers and the other unreachable peers are connected to two peers. Arrows represent connections and are directed towards the peer for whom the connection is an incoming connection.

5.2.5.4. Conclusion

Our observations of the spam wave in July and August 2021 enabled us to obtain estimates for the peer degree distribution of reachable peers and to improve on measurements of the number of reachable peers. We do not know the origin of this spam wave and the intent of the spamming entities. Interestingly, in July 2024, three years after the spam wave, CVE-2024-52919 [Tea24a] was disclosed which describes an integer overflow bug in Bitcoin Core caused by spam of ADDR messages. The fix for the issue was merged in the middle of July 2019 and released with Bitcoin Core version 22.0 in September 2019. The spam wave started in early July 2019 before the fix was merged and on June 12, 2021 there seems to have been a short test run of the spam wave which is even before CVE-2024-52919 was reported to the Bitcoin Core Security Team according to [Tea24a]. While we do not know whether there is a correlation between CVE-2024-52919 and the spam wave, the fix for CVE-2024-52919 implemented in Bitcoin Core introduced a rate limit for propagated addresses which reduces the impact of future spam waves but does not completely prevent spam waves.

5.2.6. Validation

We validate our results on the Bitcoin P2P network by integrating them in a coarse-grained model of peer connections and checking this model for plausibility. Due to the simplifications required for this modeling, the model should be interpreted only as an approximate representation. The basic idea is to estimate the total number of connections in the P2P network from the peer degree distribution and then estimate how many of these connections are created by each type of peers (we will define types of peers below). We observe that, within the model, our individual estimates are consistent which supports their plausibility and their validity.

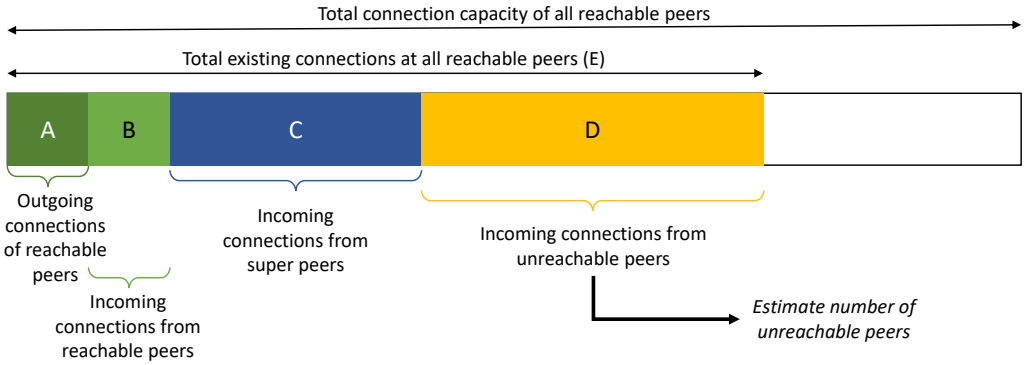

Figure 5.10.: Schematic distribution of connections at reachable peers. The colors correspond to the colors used in Fig. 5.9. Figure adapted from [GBH22].

Table 5.1.: Parameters for the coarse-grained model of connections in the Bitcoin P2P network.

Type of peers	Number of peers	Source for number of peers	Outgoing connections	Source for outgoing connections
Reachable peers	7518	own measurement	10	default in Bitcoin Core
Unreachable peers	unknown	-	9.86	measured distribution of clients and default number of outgoing connections per client
Super peers	50	own measurement	7518	own measurement of reachable peers

We assume that there are three types of peers: Reachable peers, unreachable peers, and super peers. Each of these types of peers has a distinct behavior of connecting to other peers. A simplified example of a network consisting of these types of peers is shown in Fig. 5.9. In our model, reachable peers open ten outgoing connections which corresponds to the default configuration for Bitcoin Core. Unreachable peers open connections to reachable peers. To find a plausible parameter for the number of connections opened by unreachable peers, we determine the distribution of clients used by unreachable peers by calculating the distribution of user agents that are announced to five reachable peers that we operate. Based on their distribution and the default number of outgoing connections created by each client,[5] we estimate that unreachable peers open on average 9.86 outgoing connections. In the Bitcoin P2P network, there are unreachable peers that open significantly more connections than an average unreachable peer. For example, there are peers (such as our monitor nodes) that open outgoing connections to all reachable peers. We model such peers as 'super peers'. While super peers in the actual P2P network exhibit a range of different behaviors, we model super peers as peers that open a connection to every reachable peer.

[5] *Per user agent: number of outgoing connections / share of unreachable peers.* Bitcoin Core: 10 (8 for full relay and 2 block relay only) / 86.7 %, bcoin: 8 / 5.3 %, BitcoinJ: 12 / 2.5 %, Bread: 3 / 1.1 %

In the following, we parameterize the model with the number of peers for each type of peers and, based on the number of outgoing connections created by each type of peers, we calculate the total number of connections created by each type of peers. Then, we compare the total number of connections for all types of peers to the number of connections that are inferred from the estimated peer degree distribution. A visualization of the resulting distribution of the use of connection slots at reachable peers is shown in Fig. 5.10. The parameters of the coarse-grained model as well as how we determined them are summarized in Table 5.1. For the estimates, we rely mostly on measurements made in the middle of the year 2021. We have estimated above that the number of reachable peers in August 2021 was about 7518 peers. As we assume that each reachable peer opens ten outgoing connections, there are 7518×10 outgoing connections from reachable peers (A in Fig. 5.10). Because each outgoing connection is an incoming connection at another peer, each connection between reachable peers takes up two connection slots at reachable peers. We model this as a further 7518×10 incoming connections at reachable peers (B in Fig. 5.10).

To get an estimate for the number of super peers, we use the following methodology. The idea is to determine a set of peers for which we observe a behavior that deviates from that expected for unreachable peers and then measure the average number of incoming connections from these peers to two reachable peers that are operated by us. We run five Bitcoin Core peers that accept incoming connections and log their incoming connections. Two of those five peers accept all incoming connections. The other three peers run with the default maximum of 125 connections which means that incoming connections are evicted when all slots for incoming connections are filled. We analyze the logs of these five peers for a time span of two months (August and September 2021). We define *super peers* as peers that either (1) opened connections to all five of our reachable peers on at least five days of the two months or (2) announced five or more different user agents during the two months. These two properties are unexpected for normal unreachable peers: A usual unreachable peer that opens ten connections to randomly chosen peers from the set of over 7000 reachable peers is unlikely to connect to all of our five reachable peers on the same day. If this even occurs on five of 61 days, we conclude that the unreachable peer is not a usual unreachable peer. An unreachable peer might announce different user agents during the time span of two months because the peer upgrades to a new software release. However, it is very unusual for peers to run five different software versions during a time span of two months. A reason for peers showing such a behavior is that the address that we see is a proxy used by multiple different peers. This includes, for example, Tor exit nodes which are used by Tor peers to connect to peers that are reachable over IP. Because these peers exhibit a different behavior than those peers that we model as unreachable peers, we model peers announcing five or more different user agents as super peers.

We run this analysis for the months August and September 2021 and analyze the logs of our five reachable peers. We find that from 773 addresses connections were opened to all five of our reachable peers on five or more days and that we observed from 2194 addresses at least five different user agents. There are 2375 addresses that fulfill either of the two conditions. In August 2021, two of our reachable peers have already been running for several months and were configured with the default maximum of 125 connection slots.

For these two peers, we determine that, in August and September 2021, these two peers were connected to on average 50 of the 2375 addresses. Thus, from the perspective of our peers, the peers behind the 2375 addresses exhibit a connection behavior that is equivalent to the behavior of 50 super peers that are connected to all reachable peers. Therefore, we parameterize our model with 50 super peers. It is an interesting observation on its own that on average 50 of a reachable peer's 125 incoming connections originate from peers with non-standard behavior. To stay in scope, we do not discuss this further but note that this observation is discussed by the community[6].

Because super peers connect to all reachable peers in the network, there are 50×7518 connections created by super peers (C in Fig. 5.10). To estimate the total number of filled connection slots at reachable peers (E in Fig. 5.10), we calculate the sum of all estimated peer degrees of peers that have an estimated degree of at most 130. This threshold, which is the default maximum of 125 plus a 4 % margin, excludes peers with higher degrees since they use unknown non-default configurations. Using our estimated degree distribution, we obtain an estimate of 700 541 filled connection slots at reachable peers. Subtracting the slots attributed to reachable and super peers leaves

$$700\,541 - (7518 \times 50) - 2 \times 7518 \times 10 = 174\,281$$

connection slots that are filled by unreachable peers (D in Fig. 5.10). Based on our estimate for the average of outgoing connections made by unreachable peers, we estimate that there are $174\,281/9.86 = 17\,676$ unreachable peers that are connected to reachable peers at the same time.

In the following, we relate the estimate of 17 676 unreachable peers to the measurements of the PAL method for which we measured the number of unreachable peers that announce their address during a day. Because the estimates itself are based on many assumptions whose accuracy is difficult to quantify, the calculation should be interpreted as a 'back-of-the-envelope calculation'. However, we see value in it because the calculation gives an impression of how the estimates for unreachable peers are related to each other.

The estimate of 17 676 unreachable peers from the analysis of the connection slots of reachable peers is based on observations of the spam wave in July and August 2021. Because the spamming interferes with the PAL method, we can only get an estimate from the PAL method for days at which no spam was sent. Therefore, for calculating the estimate of the PAL method, we take into account five days before the spamming, five days on which the spamming was paused, and five days after the spamming ended. Averaged over these days, the PAL method observed 27 520 unreachable peers in small ADDR messages per day (set A in Fig. 5.11). Our validation of the PAL method (see Section 5.2.4.3) showed that our monitor nodes observed on average about 95 % of addresses of unreachable peers that announced their address. Thus, we estimate that there were about $27\,520/0.95 = 28\,968$ unreachable peers that announced their address (set B in Fig. 5.11).

[6] https://bnoc.xyz/t/60-spy-peers-connected-to-freshly-setup-node/15, https://b10c.me/observations/06-linkinglion/

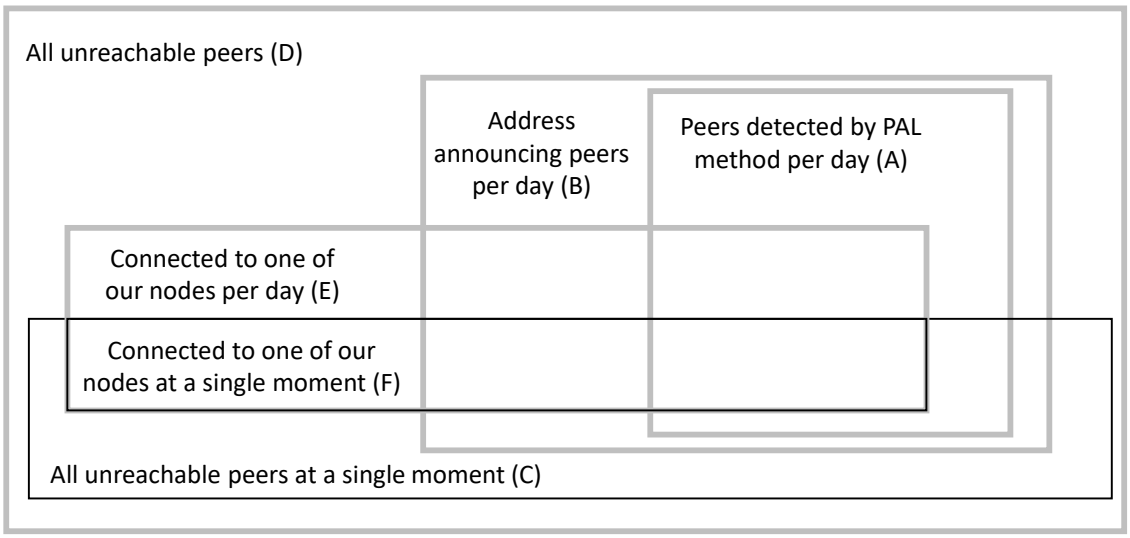

Figure 5.11.: Venn diagram for sets related to the number of unreachable peers in the Bitcoin P2P network. The largest box (D) represents the number of unreachable peers that are active on a day. Of those peers, only a subset is active at a given moment (C). Another subset of the unreachable peers per day are peers that announce their address (B). Of those peers, about 95 % are detected by the PAL method (A). Another subset of all unreachable peers active per day is connected to one of our reachable peers (E). A subset of this set are peers that are connected to one of our reachable peers at a single moment (F). Another subset of the set E are the unreachable peers that are connected to one of our reachable peers and announce their address (E ∩ B).

The estimate of 17 676 unreachable peers is an estimate for the number of unreachable peers at a single moment (set C in Fig. 5.11). This is a subset of the unreachable peers that are active on a whole day (set D in Fig. 5.11). In the following, we relate our estimate for the number of unreachable address-announcing peers per day (B) to our estimate for the number of unreachable peers occupying connection slots at a given point in time (D). To measure the expected relation of these two values, we determine the relation of address announcing peers per day to the number of all unreachable peers connected at a single moment on two of our reachable peers running Bitcoin Core with the default configuration (set E ∩ B and set F in Fig. 5.11). During August and September 2021 these two peers were connected on average to 85 unreachable peers per day that announced their address (E ∩ B). During the same time, our two reachable peers were connected on average to 45 unreachable peers (F). Thus, the relation of unreachable address-announcing peers per day to all unreachable peers at a single moment was on average $\frac{85}{45} = 1.89$. Boldly assuming that this relation generalizes to all reachable peers, we calculate based on the estimate of 28 968 unreachable address-announcing peers per day, that on average there are $\frac{28\,968}{1.89} = 15\,390$ unreachable peers at a single moment. This estimate differs about 13 % from the estimate of 17 676 unreachable peers at a single moment based on the number of occupied connection slots.

While the two estimates for the number of concurrently active unreachable peers do not align perfectly, they are still close to each other. It can be expected that there is a difference of the two estimates because we made various assumptions and simplifications for calculating the estimates. However, it is interesting to see that this difference is only

13 %. Further, the calculation shows that informally referring to 'the number of unreachable peers' leaves room for different interpretation and the calculation gives an impression of how different measurements of unreachable peers in the P2P network relate to each other.

5.2.7. Conclusion

In this section, we provided novel measurements and estimates for metrics regarding the Bitcoin P2P network. These metrics can be used to parameterize models of the Bitcoin P2P network that are, for example, used in simulations and our results have already been picked up by the developer community for this purpose [Zum23]. Still, many other metrics about the Bitcoin P2P network are unknown. While the topology of the Bitcoin P2P network is deliberately hidden because knowledge of the topology could facilitate attacks, there are arguments [FSD22] in favor of making topology information public. While it is unclear what represents a good trade-off between hiding and making information public, our results contribute to strengthening the security of the Bitcoin P2P network: The PAL method detects the number of self-announcing unreachable peers per day whose continuous measurements allow for detecting spam waves that have a security impact. The peer degree distribution of reachable peers informs the development of network models. While we run the PAL method continuously [NGH], our peer degree results are measured only for a specific time in the past. Because of the lack of more recent measurements, this result is the best that we have at the moment. Future work could explore new ways of using observations of peer behavior to learn about the Bitcoin P2P network.

5.3. Conclusion

In this chapter, we have discussed the first layer below a payment channel network. We have generalized the requirements of a first layer and created the RFL model which has reduced properties compared to a blockchain. We have shown that the RFL model suffices as first layer to implement a secure payment channel network. Further, we showed that not only Bitcoin fulfills the RFL model under certain assumptions but also central parties like banks could provide an implementation of the RFL model and thereby enable the use of payment channel networks for traditional currencies. In the second part of the chapter, we measured metrics of the Bitcoin P2P network and compared them to previous measurements. With these measurements, we contribute to a better understanding of the Bitcoin P2P network which can improve future analyses and development of the network.

Based on the findings presented in this chapter, we conclude that the assumptions that Lightning makes on Bitcoin as its underlying layer are in practice usually fulfilled. From a theoretical perspective, however, it has not yet been shown that Bitcoin fulfills the requirements of Lightning and it remains an open question of how to build a first layer that can provably implement the RFL model.

6. Extensions of and Applications for Payment Channel Networks

After we have focused at the protocol for a payment channel network in Chapters 3 and 4 and at the layer below a payment channel network in Chapter 5, we focus in this chapter on approaches that extend and use payment channel networks. We consider the research question of how payment channel networks can be applied in practice. To make payment channel networks more practical for users, we present two approaches for watchtowers for payment channels in Sections 6.2 and 6.3. Then, to explore how payment channel networks can be applied in other protocols, we present in Section 6.4 a protocol for public transport ticketing that uses a payment channel network for ticket payment to achieve a fair exchange of ticket in exchange for a payment for the ticket and to obviate the need for clearing between providers.

Payment channel users need to watch the blockchain for outdated commitment transactions published by the other party. The interval how often a user needs to check the blockchain can be chosen when a channel is opened. For this choice, there is a trade-off: The longer the interval, the less frequently a user has to check the blockchain. However, the longer the interval, the longer users need to wait until they can redeem their coins if they close a payment channel by publishing a commitment transaction. The idea of watchtowers is that, if a user wants to go offline for an extended period of time, the user can outsource the task of watching and, if needed, punishing to a third party called a watchtower. Thereby, users can choose lower intervals even if they might be offline for periods longer than the interval. We present two approaches for watchtowers. TEE Guard is an approach that reduces the storage requirements of watchtowers by using Trusted Execution Environments (TEEs) to store secrets at watchtowers while preventing them from misusing the secrets. A second watchtower approach decouples storing data and watching the blockchain using the InterPlanetary File System (IPFS). These watchtower approaches contribute to rendering the use of payment channel networks practically feasible for end users.

We explore the use of payment channel networks in an application protocol at the exemplary use case of ticketing in public transport. We present a decentralized ticketing protocol for public transport that is built on a payment channel network. The protocol allows customers to register once with a host provider and to buy tickets from any provider. By using a payment channel network for ticket payment, the protocol achieves two useful properties: First, the protocol implements that the selling provider is paid directly without centralized fare clearing. Second, ticket purchase is implemented by a fair exchange of a

ticket against the payment for the ticket which ensures that each ticket is paid for and that a customer receives a valid ticket after having paid the ticket. We find that the use case shows that payment channel networks provide a useful component to build practical protocols.

TEE Guard was previously published in [Lei+19], the watchtower approach using IPFS was published in [BG22], and the ticketing protocol using payment channel networks was published in [GvZH22].

We start the section by giving an overview of other watchtower approaches and we introduce TEEs and IPFS in Section 6.1. We present the two watchtower approaches in Sections 6.2 and 6.3. The ticketing protocol is introduced in Section 6.4. Finally, we conclude in Section 6.5.

6.1. Fundamentals and Related Work

In this section, we complement the fundamentals and related work that was discussed in Chapter 2 by work that is relevant in particular for Chapter 6. We first give a more detailed overview of watchtower approaches in Section 6.1.1. Then we introduce Trusted Execution Environments (TEEs) in Section 6.1.2 and the InterPlanetary File System (IPFS) in Section 6.1.3. We will use TEEs and IPFS for the watchtower approaches that we propose in this chapter.

6.1.1. Watchtower Approaches

Baseline Watchtower Approach The first proposal for a watchtower was presented by Dryja in 2016 [Dry16]. The idea of the approach is that a customer, i.e., a user who employs a watchtower, sends to the watchtower a revocation transaction for every outdated commitment transaction of a channel. To avoid that the watchtower can observe all updates in the channel, the revocation transactions that are sent to the watchtower are encrypted. The key for the encryption is derived from the id of the outdated commitment transaction that is spent by the revocation transaction. If an outdated commitment transaction is published on the blockchain, the watchtower can derive the encryption key, and decrypt and publish the revocation transaction.

Besides the frequency of updates, the watchtower does not learn any information about updates of the payment channel because the watchtower sees only those transactions in plaintext that are published on the blockchain. There are three main drawbacks of Dryja's approach that relate to storage, accountability, and incentivization. A first drawback is that the watchtower requires storage for the encrypted revocation transactions that is linear in the number of updates in the channel. A second drawback is that the approach does not hold a watchtower accountable if the watchtower does not react to an outdated commitment transaction. And third, the approach assumes that a watchtower is altruistic,

i.e., there is no mechanism to incentivize the watchtower such as payments from the customer.

A refined version of Dryja's approach has been implemented in Rust [Lab26] under the name The Eye of Satoshi (TEOS). This implementation includes a mechanism from [McC+19] (introduced below) for paying the watchtower and for holding the watchtower accountable if the watchtower fails to react to an outdated commitment transaction. A further variant of Dryja's approach is implemented as part of the Lightning implementation LND [Lig].

Improved Watchtower Approaches The drawbacks of accountability, incentivization, and storage have been addressed by other approaches for watchtowers including the two approaches that we present in this chapter. In the following, we briefly present approaches of other authors. Pisa [McC+19] is a watchtower approach that targets accountability by requiring the watchtower to deposit funds as a security. During each update, the customer receives a signed receipt from the watchtower. If the watchtower later fails to revoke an outdated commitment transaction, the signed receipt can be used to prove that the watchtower was responsible for revoking the commitment transaction and the watchtower loses the deposited funds.

A different approach for incentivization is implemented by DCWC* [Ava+18b]. In this approach, each revocation transaction contains an additional output that pays the watchtower. Thereby, a watchtower is incentivized to publish a revocation transaction if an outdated commitment transaction is published on the blockchain. Cerberus Channels [ATW20], an alternative payment channel protocol, uses a further approach for incentivization which includes a payment to the watchtower for every update of the channel combined with a security deposit of the watchtower as used by Pisa.

A theoretically interesting idea for reducing a watchtower's storage requirements is the Outpost [KNW19] approach. The idea of this approach is to modify commitment transactions so that they contain the associated encrypted revocation transactions. Thereby, the watchtower just needs to store the decryption keys to be able to extract the revocation transaction from an outdated commitment transaction and publish the revocation transaction. The drawback of this approach is that the approach significantly increases the size of commitment transactions and, thus, the fees associated with publishing a commitment transaction even if the latest commitment transaction is published by an honest user. A variant of this idea has been reused in the Garrison [Mir+22] approach.

A further approach [LSS20] integrates watchtowers into the closing procedure of a payment channel. If a watchtower is unavailable, a channel can be closed only after an especially long waiting time. Otherwise, it is ensured that the closing state of a payment channel is at least as new as the state stored by the watchtower. A further approach to improve the availability of a watchtower is to distribute the task of a watchtower from a single party to a group of parties (see SilenTower [XZZ24]).

Protection of HTLCs in Approaches Based on Sharing Revocation Transactions A fundamental limitation of the approach of sharing revocation transactions with the watchtower is that is is hard to cover outputs of HTLCs. This applies not only to the baseline approach but also to most of the improved approaches. A simple approach to cover outputs of HTLCs would be that the customer creates, for each commitment transaction, a revocation transaction that spends all outputs of the commitment transaction and sends the funds to the customer. However, if a commitment transaction contains an HTLC, this approach is not sufficient because the watchtower needs two revocation transactions for the following two scenarios: (1) In the simplest case, the commitment transaction with the HTLC is published and the watchtower can publish the regular revocation transaction. The revocation transaction spends all outputs of the commitment transaction. (2) It might be that the commitment transaction with the HTLC is published and the other party in the channel publishes an HTLC success or HTLC timeout transaction. In this case, the regular revocation transaction that spends all outputs of the commitment transaction is invalid because the HTLC output of the commitment transaction is already spent. Thus, a second revocation transaction is needed that spends only the non-HTLC outputs of the commitment transaction and the output of the HTLC success or HTLC timeout transaction. Thus, to cover a commitment transaction with one HTLC, two different revocation transactions need to be stored by the watchtower. The number of required revocation transactions increases exponentially with the number of concurrent HTLCs. Thus, implementations of watchtower approaches that are based on sharing revocation transactions with the watchtower like TEOS [Lab26] and LND [Lig] do not cover outputs of HTLCs. Consequently, even users who employ one of these watchtowers remain exposed to potential losses of up to the cumulative value of all HTLCs that were simultaneously active in their payment channels at any given time in the past. With the watchtower approach based on TEEs that we introduce in Section 6.2, we present an approach that can also protect the amounts of HTLCs.

6.1.2. Trusted Execution Environments

We introduce Trusted Execution Environments (TEEs) because one of the watchtower approaches that we present in Section 6.2 uses TEEs. A TEE is a secure environment within a processor. A TEE protects the confidentiality and integrity of the code executed in the environment (*isolated execution*) and of the data that is used (*protected memory*). A TEE is based on features built into the hardware and firmware of a processor that protect the code and data even from the operating system. While there are different implementations of TEEs, we assume in the following a TEE that offers a feature set that is comparable to Intel SGX. In particular, we require that that the TEE offers isolated execution, protected memory, and remote attestation. Using *remote attestation*, the integrity of the hardware on which the TEE is running can be verified. Further, it can be verified that a specific piece of code is executed inside the TEE. In Intel SGX, the attesting entity needs to trust Intel, the hardware manufacturer, to verify the attestation. In Intel SGX, the secrets used for the remote attestation can be reused to establish an authenticated and

encrypted communication channel between the TEE and the entity that requested the remote attestation.

6.1.3. IPFS

The second watchtower approach that we present in Section 6.3 uses the InterPlanetary File System (IPFS) [Ben14]. IPFS is a decentralized file system that is implemented as peer-to-peer network. Given the address of a file, IPFS can be used to find a peer that the file can be downloaded from. In IPFS, files are addressed using a content identifier (CID) that is derived from the file's content through a cryptographic hash function. Consequently, a file's address depends on the file's content and any modification to the file's content results in a new CID. This property represents a challenge when a file must be shared via its CID but must also remain modifiable. To address this challenge, IPFS integrates the InterPlanetary Name System (IPNS).

IPNS resolves the problem of assigning a persistent identifier to IPFS files even if they are modified. An IPNS address is a dynamic address that can be updated to reference new CIDs. To create an IPNS address, an asymmetric key pair is generated where the hash of the public key serves as the IPNS address. The corresponding private key is used to sign address records which specify the current CID associated with the IPNS address. Thereby, IPNS provides mutable references to IPFS files. If a file is shared via an IPNS address, the file can be modified and the owner of the IPNS address can update the address record to reference the new CID.

IPFS allows for representing directories that can contain files or other directories. A directory is a special file that consists of a list of file names of which each file name is associated with a CID that references a file or a CID of a subdirectory. A directory also has a CID that is based on the directory's contents.

IPFS provides a mechanism to find files but does not ensure that a file is accessible. For files to remain accessible, they must be hosted by at least one peer in the network. Permanently hosting an IPFS file is called pinning. Since peers operated by individual users may not be continuously online, local pinning at a user's own hardware alone is insufficient for guaranteeing availability. To increase the availability of a file, third-party services offer to pin files in exchange for a fee. Examples for such pinning services include Pinata[1], Filebase[2] and nft.storage[3].

[1] `https://pinata.cloud`
[2] `https://filebase.com`
[3] `https://nft.storage`

6.2. Watchtower Approach using Trusted Execution Environments

The watchtower approach that we present in this section tackles two main challenges of a watchtower: storage requirements and availability. Further, it also protects the amounts of HTLC outputs in a commitment transaction without additional complexity.

A main challenge for a watchtower approach is the amount of data that needs to be stored by the watchtower. This has the following background: As introduced in Chapter 2, each new commitment transaction in a payment channel uses a new revocation key. Because each previous commitment transaction could be published on the blockchain, the users of a payment channel need to store the revocation keys for all existing commitment transactions. Lightning uses a key derivation approach for revocation keys that allows to store only secrets of a size that is logarithmic in the number of updates. If an outdated commitment transaction is observed on the blockchain, a user will derive the corresponding revocation secret and create, sign, and publish the revocation transaction. In a watchtower approach such as the basic approach by Dryja [Dry16] (see Section 6.1), the watchtower is untrusted. Therefore, a customer of a watchtower does not share the revocation secrets with the watchtower because, in case an outdated commitment transaction has been published on the blockchain, the watchtower might use the secrets to create a revocation transaction that sends the channel's funds to the watchtower instead of to the customer. Instead of sharing revocation secrets, the customer shares pre-signed revocation transactions with the watchtower that send the channel's funds to the customer in case of fraud. However, these revocation transactions cannot be stored as efficiently as the secrets used to derive the revocation keys for signing revocation transactions.

The idea of the watchtower approach that we present in this section is to share the revocation secrets with the watchtower but to use a TEE to restrict the watchtower's use of these secrets. The TEE protects the secrets so that they can be only read and processed with predefined code that uses the secrets only to create revocation transactions that send a channel's funds to the customer. This approach greatly reduces the watchtower's storage requirements and thereby reduces the cost for running a watchtower.

If the customer of a watchtower goes offline, the customer relies on the availability of the watchtower, i.e., that the watchtower continues to watch the blockchain and reacts to outdated commitment transactions. However, it might be that a watchtower is employed by a customer but stops its service temporarily or permanently without notifying the customer. To increase the availability, the TEE Guard approach assumes a distributed network of independent watchtowers. A customer employs multiple watchtowers and, to protect the customer, it suffices if one of the watchtowers reacts to outdated commitment transactions. Further, to incentivize watchtowers to stay available, the watchtowers in TEE Guard regularly check that the other watchtowers are available. Based on the results of these regular checks, irregularities with the availability of watchtowers can be discovered by customers.

We developed the TEE Guard watchtower approach together with Marc Leinweber, Leonard Schönborn, and Hannes Hartenstein. A detailed description of this approach can be found in the master thesis of Leonard Schönborn [Sch19] and in [Lei+19].

This section is structured as follows. We give an system overview and introduce the design goals of the watchtower approach in Section 6.2.1. We present the architecture and describe the operation in Section 6.2.2. We discuss and evaluate the approach in Section 6.2.3 before we conclude the section in Section 6.2.4.

6.2.1. System Overview and Design Goals

System Overview In the TEE Guard approach, there are multiple independent watchtower providers. Each watchtower provider runs a set of watchtower instances. Users can become customers of watchtower providers. Each customer is assigned a unique watchtower instance that is responsible for watching the customer's channels. A customer pays the customer's watchtower provider on a regular basis (e.g., monthly). Typically, a user will be customer at multiple watchtower providers so that the user's payment channels are protected even if one watchtower provider fails. We assume that watchtower providers might be malicious which includes that they might stop their watching service although they are being paid or that they might try to use their knowledge to steal a user's funds. The network connection between the watchtower providers with each other and to the blockchain is untrusted and unreliable (Dolev-Yao model). However, watchtower providers care about their reputation. Therefore, they do not misbehave if others could prove their misbehavior afterwards. We assume adversarial users cannot break cryptographic primitives or TEEs. We discuss the impact of attacks on TEEs in Section 6.2.3.5.

Design Goals The design goals for the distributed watchtower system are as follows: Users should be able to rely on the watchtowers to be available, to monitor the blockchain, and to publish revocation transactions when needed. For watchtower providers, the approach should scale to many channels that can be watched with a single watchtower instance. Further, the cost to run a watchtower should be low enough so that a watchtower platform can be economically run if users pay reasonably high fees. Information about the monitored channels such as the identity of the counterparty, intermediate balances, and direction of payments should stay private to the two parties that are involved in the payment channel. However, we accept that the watchtower provider can learn how often a customer updates one of their payment channels by observing traffic patterns.

Quantification of Availability Requirement Due to maintenance or temporary network outages, a watchtower might not be always available. Such outages are acceptable if the overall availability is high enough. We quantify the availability requirement based on the idea that the expected outcome of cheating should be negative for the cheating party. If one of the parties of the channel cheats by publishing an outdated commitment transaction,

the party can gain at most the difference between the party's latest balance and the party's balance in the state in which the cheating party had the lowest balance. To ensure that each party has something to lose if the party cheats, each channel in Lightning is parameterized with a *channel reserve*. BOLT 2 [Lig26a] suggests to use a value of 1 % of a channel's capacity and lnd and Core Lightning, two widely used [MZ21b, Appendix A] implementations of Lightning, use that value as default. In the following, we assume that channels use this value for the channel reserve. When sending a payment, each party has to keep at least the channel reserve to ensure that the party has something to lose. Lightning ensures that no party can send a payment if the party's balance after the payment would be lower than the channel reserve. A special case is that, when a channel is opened, the funder typically owns the total balance in a channel and, thus, the fundee (the counterparty of the funder) has initially less than the channel reserve. Consequently, the maximum value that the channel's funder can gain by cheating is, if the funder's balance has decreased to the channel reserve, the difference between the channel's capacity and the channel reserve, i.e. 99 % of the channel's capacity. The maximum value that the fundee can gain is lower because there cannot be a state in which the fundee has less than the channel reserve. Thus, we only consider the case for the funder as a worst case scenario. Let p be the probability of a successful cheating attempt of the funder, i.e. the funder publishes an outdated commitment transaction and the fundee does not react within the necessary time window. In the worst case scenario, the funder's balance is only the channel reserve and the funder closes the channel with the initial commitment transaction that sends all of the channel's funds to the funder. If the cheating is detected and the funder is punished, the funder loses the channel reserve, i.e. 1 % of the channel's capacity. If the cheating is undetected, the funder wins 99 % of the channel's capacity. Thus, the expected outcome of the funder is as follows where p is the probability that a cheating attempt is successful:

$$e = p \times -1\,\% + (1 - p) \times 99\,\%$$

If the expected outcome is positive, the funder is economically incentivized to attempt the cheating. To make a cheating attempt non-profitable, it must hold that the expected outcome is negative:

$$\begin{aligned} 0 > e &= p \times -1\,\% + (1 - p) \times 99\,\% \\ &= p \times -1\,\% + 99\,\% - p \times 99\,\% \\ &= -p + 99\,\% \\ \Rightarrow \quad p &> 99\,\% \end{aligned}$$

If the probability to detect and punish a cheating attempt is above 99 %, cheating is unprofitable with a channel reserve of 1 %. Thus, we aim to achieve an availability of at least 99 % with the watchtower approach.

6.2.2. Architecture and Operation

Figure 6.1 gives an overview of the TEE Guard architecture. Users who have payment channels can become customers of watchtower providers. Each watchtower provider

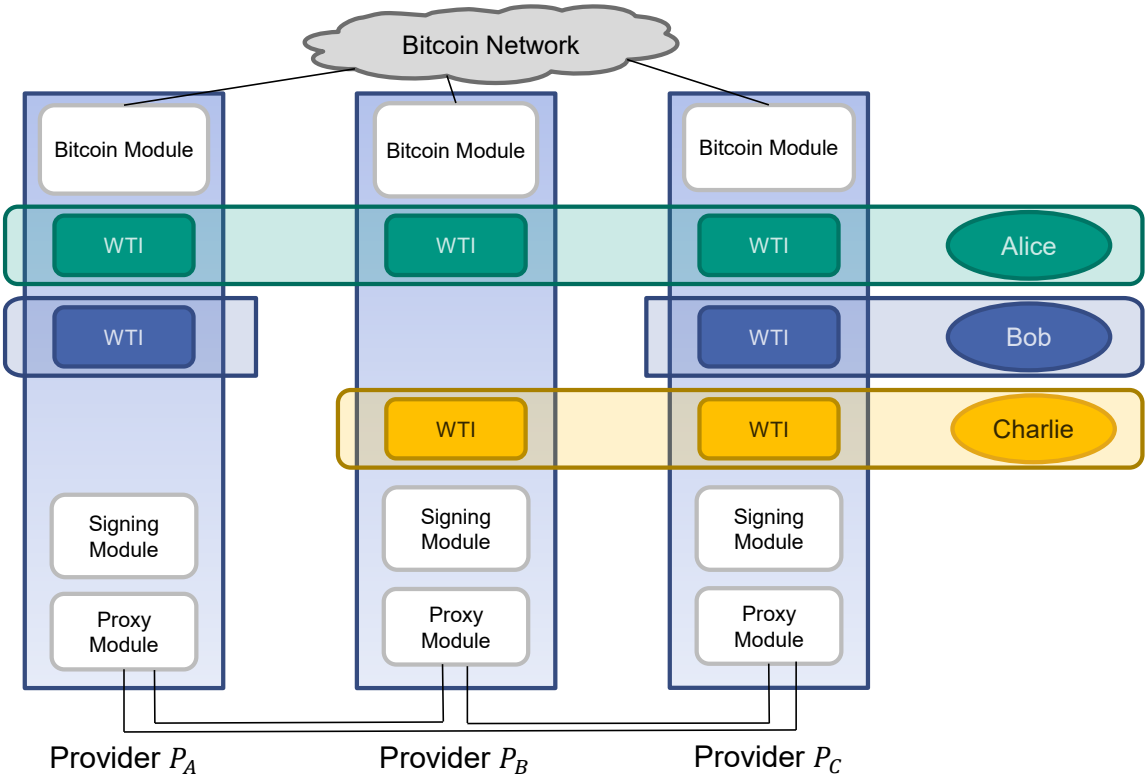

Figure 6.1.: Overview of the TEE Guard approach for watchtowers. An example is shown with three watchtower providers P_A, P_B, P_C and three users Alice, Bob, and Charlie. Each watchtower provider runs a bitcoin module, several watchtower instances (WTI), a signing module, and a proxy module. All modules are run inside a single TEE per provider platform. Customers can have watchtower instances at multiple providers. Figure taken from [Lei+19] (redacted). Reproduced with permission from Springer Nature.

runs a watchtower platform that consists of a TEE that contains multiple modules. The modules run in the same TEE and are only logically separated. A watchtower provider runs multiple watchtower modules, called watchtower instances, one for each customer. Each watchtower provider runs a bitcoin module that connects to the peers in the Bitcoin P2P network to retrieve blocks and publish transactions. Customers who have watchtower instances at multiple watchtower providers, can connect their watchtower instances. Connected watchtower instances exchange information about the watched channels to ensure consistency between the connected instances. At each provider, the connections between watchtower instances are tunneled through a proxy module to allow for using the same connection between two providers for multiple connected watchtower instances. Two watchtower providers that host connected watchtower instances monitor each other's availability and create availability logs. Each watchtower provider runs a signing module that signs these availability logs. The signed availability logs are published and customers can inspect them to verify that the watchtower providers that they have hired have actively monitored the blockchain.

6.2.2.1. Setup

To allow for customers to inspect and verify the code, TEE Guard should be open source. A watchtower provider starts a watchtower platform by building and starting the TEE Guard software. While a watchtower provider might modify the software's source code, the remote attestation of the TEE will be used to ensure that the correct code is running inside the TEE.

If a customer requests a watchtower instance, the provider instructs the platform to create a new watchtower instance and gives the customer access to the watchtower instance. Using remote attestation, the customer establishes a secure channel to the watchtower platform and verifies that the watchtower platform runs the expected code. The customer identifies by a `revocation_payout_address` which is the address that a revocation transaction should send the funds to.

To connect two watchtower instances A and B that run at two different providers P_A and P_B, the customer instructs one of the watchtower instances (say, A) to connect to the other watchtower instance (B). If the proxy modules of the respective watchtower platforms of providers P_A and P_B are not yet connected, the proxy module of provider P_A will connect to the proxy module of provider P_B and both proxy modules will perform a mutual remote attestation. The proxy module of provider P_B searches for the watchtower instance B run by provider P_A by looking for a watchtower instance that has the same `revocation_payout_address` as watchtower instance A. Finally, the connection between watchtower instances A and B is established by storing inside the watchtower instances that they are connected to each other and by storing the required routing information inside the proxy modules of provider P_A and provider P_B.

6.2.2.2. Operation

Each watchtower instance stores a list of monitored channels and the information about the monitored channels that is required to monitor the blockchain for outdated commitment transactions and to create revocation transactions if necessary. Table 6.1 lists the required data: First, a field is stored that specifies the funding output by referring to the funding transaction and the respective output number. If a commitment transaction is found that spends this output, the watchtower needs to check whether the commitment transaction is the most current commitment transaction. To this end, the most current commitment number is stored which can be compared to the commitment number included in a published commitment transaction. The commitment number included in a commitment transaction is encrypted using a key derived from both channel parties' payment basepoints. To decrypt the commitment number, the watchtower instance stores both channel participant's payment basepoints. If a published commitment transaction is outdated, the watchtower instance needs to know the necessary information to create a revocation transaction. Therefore, the customer's revocation basepoint secret and the root values to derive the `per_commitment_secrets` are stored.

Table 6.1.: Information required for watchtower operation that has to be stored for a channel by each watchtower

Data	Purpose
`funding_output`	identification of channel-related commitment transactions
both participants' `payment_basepoints`	decryption of `commitment_numbers` in commitment transactions
most current `commitment_number`	identification of outdated commitment transactions
`revocation_basepoint_secret`, values to derive `per_commitment_secrets`	revocation key derivation

If a customer updates their channel, the customer sends an update to the watchtower instance that includes the updated commitment number as well as the `per_commitment_secret` for the outdated transaction. The `per_commitment_secret` is stored by updating a root value that can be used to derive all known `per_commitment_secrets` (see [Lig26b]). Each watchtower provider has a state counter in the proxy module that is incremented when the list of monitored channels of a watchtower instance is updated or when a connection to a watchtower instance on another watchtower platform is added or removed.

The blockchain module connects to a set of peers in the Bitcoin P2P network. If one of the connected peers announces a new block, the blockchain module downloads the block and verifies that the block is a valid Bitcoin block. The blockchain module inspects the transactions contained inside the block. If an output is spent that is on the list of monitored channels, the blockchain module invokes the corresponding watchtower instance. The watchtower instance reads the respective commitment transaction and decrypts the commitment number contained in the transaction. If the commitment number is smaller than the latest commitment transaction stored in the watchtower instance, the watchtower instance calculates the corresponding revocation key based on the `revocation_basepoint_secret` and the `per_commitment_secret`. If the revocation key is not valid for signing a transaction that spends the outputs of the commitment transaction, the commitment transaction was published by the customer of the watchtower and the watchtower cannot revoke the transaction. However, if the commitment transaction was published by the counterparty of the customer of the watchtower, the watchtower instance can create a revocation transaction that spends all outputs of the commitment transaction and sends the funds to the `revocation_payout_address`. The watchtower instance sends the revocation transaction to the blockchain module which publishes the transaction by sending the transaction to the connected Bitcoin peers. To ensure that the watchtower provider cannot prevent a specific revocation transaction from being published without preventing blockchain communication at all, the communication between the blockchain module and

the Bitcoin peers is encrypted[4]. After having published the revocation transaction, the blockchain module monitors the following block to ensure that the revocation transaction is confirmed by being included in a block. If the revocation transaction is not included within the next blocks, it might be that a conflicting HTLC second-stage transaction was published. In this case, a new revocation transaction is created that spends the output of the HTLC second-stage transaction. A further reason for the revocation transaction not being confirmed is that the fees paid by the transaction are too low. In this case, a revocation transaction is created that pays more fees.

NEW BLOCK Messages For Availability Monitoring and Watchtower Synchronization After a new block has been processed by the blockchain module and, if necessary, revocation transactions have been created by the watchtower instances, the blockchain module triggers the proxy module to send a NEW BLOCK message to all connected provider platforms. The NEW BLOCK messages are used for creating the logs for checking a watchtower's availability, for synchronizing the state of connected watchtower instances, and for protection against rollbacks (see below). The NEW BLOCK message contains

- the blockchain height of the block that has just been processed,
- the current state counter, and
- the current commitment number of each monitored channel of each watchtower instance that is connected to a watchtower instance on the provider platform to which the NEW BLOCK message is sent.

When a NEW BLOCK message is received, the sender and the sender's blockchain height are logged to enable an analysis of continuous availability of the provider platform. When a NEW BLOCK message is sent, the local blockchain height is appended to the availability log. Further, when a NEW BLOCK message is received, the current state counter is logged to enable a distributed rollback protection which we explain below. To ensure that each watchtower instance has the most recent data about the monitored channels, the data of monitored channels is synchronized with each NEW BLOCK message. If a NEW BLOCK message is received, the contained data is compared with the locally stored data. If the received commitment number for a monitored channel is larger than the locally stored commitment number, the respective PER_COMMITMENT_SECRETS are requested from the other watchtower provider to update the local state.

Backup of TEEs and Rollback Protection TEEs protect the data in memory and can export their data in encrypted and authenticated form. This export is referred to as sealing. The challenge of the functionality of sealing data and loading the sealed data is that the TEE can be rolled back to an earlier state. Such a rollback could be used to revert recent changes

[4] Encrypted communication between peers of the Bitcoin P2P network has been proposed in BIP 324 [Meh+24] and is implemented in Bitcoin Core and enabled by default since Bitcoin Core version 27.0 which was released in April 2024.

in the state of the TEE. One way to prevent such rollback attacks is to use monotonic hardware counters in non-volatile memory that are incremented with each state change. The TEE enforces that a sealed state can only be loaded if the values of the hardware counters in the sealed state match those stored in the hardware. The problem with this approach is that hardware counters are relatively slow and allow only for a limited number of writes [Mat+17]. An alternative approach is to use a distributed counter that is stored in other TEEs that are connected in a distributed system [Mat+17]. The basic idea of the approach is that before each state change a local counter is incremented and then a byzantine atomic broadcast protocol is used to distribute the counter value to other TEEs in the distributed system. The counter value is included in each sealed state. When a sealed state is loaded, the counter values at the remote TEEs are requested. A rollback is detected and loading the state is aborted if the largest value retrieved in the responses is larger than the counter's value in the sealed state. Otherwise, the TEE resumes normal operation from the loaded state. We use this approach in TEE Guard by leveraging the connected provider platforms to remotely store the counter values of each provider platform. The counter is incremented when the set of monitored channels by a watchtower instance is changed or when a watchtower instance is connected or disconnected. However, the counter is not incremented when a channel is updated to a new commitment number. The reason for this is that incrementing the counter limits the performance of state updates and is, thus, not used for channel updates which occur relatively frequently. Consequently, the freshness of the connected watchtower instances and the set of monitored channels is ensured but the most recent updates of channels might be lost.

Payment of Watchtower Provider We assume that a customer pays employed watchtower providers in regular time intervals. We leave it open how the payment is implemented. One approach would be to use a unidirectional payment channel from the customer to the watchtower provider. A unidirectional channel has the advantage for the customer that the customer does not need to regularly watch the blockchain because old states in such a channel would be only favorable for the customer but not for the watchtower provider.

Verification of Watchtower Availability To verify that a watchtower provider has been running continuously and monitored the blockchain, users can inspect and analyze the availability logs created by each provider. More specifically, users verify that a watchtower provider was running continuously and that the network connections to the blockchain as well as all connected watchtower providers were working. Each availability log contains the sender of each received NEW BLOCK message as well as the blockchain height of the sender's most recent block. Additionally, the log contains an entry for each sent NEW BLOCK message that contains the blockchain height that was included in the sent message. Each availability log is signed by the TEE so that users can verify the log's authenticity. Users verify that a watchtower has been running continuously by checking that the blockchain heights in the log are monotonically increasing. If there is an entry in the availability log for which the blockchain height is lower than a previous entry, either

the watchtower platform that sent the NEW BLOCK message or the watchtower platform that created the log was temporarily unavailable. In general, it cannot be concluded which of the watchtowers was unavailable or whether a network connection between two watchtower providers was faulty. From the availability logs, it can be observed how long a period of unavailability lasted. It is acceptable if a watchtower was unavailable only for one or two blocks. However, if a watchtower is unavailable for longer periods of time, the signed logs damage the watchtower's reputation. Thus, we expect that watchtowers will strive for achieving a high availability.

6.2.3. Evaluation

We evaluate the TEE Guard approach with regard to reliability, scalability, deployability, and confidentiality. Finally, we evaluate the effect of attacks on TEEs on TEE Guard.

6.2.3.1. Reliability

The main idea in TEE Guard for achieving a high reliability is that a user hires multiple independent watchtowers. By assuming that the watchtowers are independent, we can calculate the availability p_n of n independent watchtowers based on the probability p_1 that a single watchtower is available:

$$p_n = 1 - (1 - p_1)^n$$

Even for a low availability of a single watchtower of $p_1 = 80\%$, a small number of $n = 3$ independent watchtowers achieves an aggregated availability of 99.2 %. This is sufficient to fulfill our goal of a 99 % availability to make the expected outcome of a cheating attempt negative (see Section 6.2.1). Thus, for the following analysis, we assume that each user hires three watchtowers.

TEE Guard makes the actual availability of watchtowers transparent using availability logs in which watchtowers create an entry when they receive have processed a new block or when they have received a NEW BLOCK message from another watchtower. Users can inspect these logs to more accurately estimate each watchtower's availability so that they can tune the parameter for the number of hired watchtowers n if necessary. If a watchtower is not available after a new block is created, this unavailability is detectable for the following reason: In such a case, the NEW BLOCK message of the watchtower will either not appear in other watchtowers' availability logs or will be sorted after entries with higher block heights. Note that it can happen that two blocks are created within a very short time window so that computation and network delays lead to a non-monotonic sorting of availability logs. To prevent a watchtower from being falsely flagged as unavailable in such a case, such cases with a short delay between two consecutive blocks can be detected by inspecting the timestamps that are included in blocks and ignored.

Table 6.2.: Memory requirements of a provider platform

Data	Size
per connected watchtower:	83 byte
IP address and port	18 byte
ECDH public key	33 byte
symmetric key	32 byte
per customer:	119 byte
`revocation_payout_address`	32 byte
index list of connected watchtowers	$n_{\mathrm{E}} \times 4$ byte
key material for communication:	87 byte
ECDH public/private key	65 byte
symmetric key	32 byte
per monitored channel:	1998 byte
`funding_txid`	32 byte
index of funding output	2 byte
`commitment_number`	6 byte
both participants' `payment_basepoints`	2×32 byte
`revocation_basepoint_secret`	32 byte
`per_commitment_secrets`	49×38 byte

6.2.3.2. Scalability

To evaluate the scalability of the approach, we analyze the resource requirements of a provider platform with respect to memory, disk space, network bandwidth, and computation. For the evaluation, we use two scenarios for which we evaluate the resource requirements and we estimate the associated cost for the watchtower. The first scenario is a low-use scenario; the second scenario is a scenario with high usage. In the first scenario, a watchtower is hired by 100 users who hire the watchtower on average to monitor three channels. The second scenario is that a watchtower is hired for every channel in the Lightning Network as of February 2026. According to public monitoring sites[5], the Lightning Network consists of about 40 000 payment channels between 17 300 nodes as of February 2026. Thus, in the second scenario, we assume 17 300 users that hire the watchtower to monitor in total 40 000 payment channels (on average, 2.3 channels per user).

In the following, we use n_{U} for the number of users that hired the watchtower, n_{C} for the total number of channels monitored by the watchtower, n_{P} for the total number of provider platforms, and n_{E} for the average number of watchtower instances per user.

[5] `https://mempool.space/lightning`

Memory For the memory analysis, we consider only the memory requirements that scale with the number of customers and the number of monitored channels. The remaining memory requirements are not significant because we assume that the number of provider platforms and the number of watchtower instances per user is significantly lower than the number of users. In Table 6.2, we list the memory requirements of the TEE per customer and per monitored channel. A watchtower instance needs 119 byte for customer that is connected to the watchtower instance and 1998 byte to store that data of each monitored channel. It follows that the memory requirement of the TEE of a watchtower platform is:

$$n_{\mathrm{U}} \times 119\,\mathrm{byte} + n_{\mathrm{C}} \times 1998\,\mathrm{byte}$$

Applying this formula to the first scenario with 100 customers and 300 channels, we estimate that a watchtower needs about 600 kB of memory. In the second scenario with 40 000 channels and 17 300 users, a watchtower needs about 78 MB of memory. We conclude that for the memory requirements, even the second scenario is practical. While in the first version of SGX the usable memory was limited to less than 100 MB, the newer SGX2 does not have this limitation anymore [McK+16]. Because revocation secrets can be securely stored inside the TEE, TEE Guard can store revocation secrets inside the TEE and thereby scale to a high number of channels independent of the number of updates in the monitored channels.

Disk Space A watchtower stores on disk the latest blocks of the blockchain. Because, to minimize the risk of blockchain forks, the transactions in a block are seen as confirmed only after the block has about six successors, TEE Guard stores the latest six block on the disk. The size of the availability logs depends on the number of connected watchtower platforms. For each new block, the log contains the current height of the blockchain and an identifier for the watchtower itself and each connected watchtower. If a watchtower is connected to 100 other providers, the required disk space for the availability log is about 40 MB for one year for the average block generation rate of Bitcoin of one block every ten minutes. Thus, we deem the required disk space to be moderate.

Computation The computation requirements for TEE Guard are also moderate. During typical operation, each transaction in each new block needs to be compared to the list of funding transaction ids. If a commitment transaction is found that closes a monitored channel, more computation is needed because, if the commitment transaction is outdated, a revocation transaction needs to be created and signed. However, channel closures are expected to occur only infrequently and new blocks arrive on average only every ten minutes so that computation does not need to be very fast. If a channel is updated, the watchtower simply stores the received data. Thus, the computation requirement is moderate.

Network The most important factors for the required network bandwidth are the reception of new blocks and channel updates of users. New blocks are expected to be in the

range of a few megabytes per hour. To that adds the size of the NEW BLOCK messages that are exchanged between the watchtowers for each new block. The NEW BLOCK messages include the current commitment numbers of all payment channels that are monitored by the sender and the receiver of the NEW BLOCK message. In the second scenario in which 40 000 channels are monitored, this results in approximately 234 kB per NEW BLOCK message if the sender and the receiver monitor all 40 000 channels. Because these messages are exchanged on average only every ten minutes, even assuming 100 connected provider platforms, a bandwidth of 1 Mbit/s is sufficient for processing new blocks. For each update of a channel, a customer has to transmit about 75 byte to the watchtower. Therefore, the required bandwidth for channel updates can be estimated as 75 byte $\times$ $n_C \times \beta$ where n_C is the total number of channels monitored by the watchtower and β is the frequency of channel updates. Assuming 40 000 monitored channels and a high average channel update frequency of one update per second ($\beta = \frac{1}{s}$), the required bandwidth for channel updates is less than 23 Mbit/s which is a moderate requirement for a professionally hosted system. This bandwidth adds up to a monthly traffic of about 7 TB per month.

6.2.3.3. Deployability

We use the estimates from the previous section to calculate the cost of operating a watchtower in the Azure Cloud. To this end, we use the Azure pricing calculator[6] to estimate the monthly cost as of March 2026 for a system hosted in Germany. For the requirements for the first scenario with 100 users and 300 monitored channels, the monthly cost is about 60 USD. For the requirements for the second scenario with 17 300 users and 40 000 monitored channels, the monthly cost is about 500 USD. Thus, the monthly cost per user is 0.6 USD in the first scenario and 0.03 USD in the second scenario. We conclude that even if a watchtower is not used to capacity as in the first scenario, the monthly cost of 0.6 USD for watching three channels per user is reasonable and, from an economical perspective, the TEE Guard approach is deployable.

6.2.3.4. Confidentiality

Per the assumption for TEEs, all secrets handled by the watchtower are kept confidential even for the entity operating the watchtower server. All data stored in memory as well as on hard disk is encrypted and, thus, not visible to the operator. If customers send secrets to a watchtower, they use encrypted and authenticated channels which are established during remote attestation. Thereby, secrets are protected by the TEE during upload, use, and storage.

[6] `https://azure.microsoft.com/en-us/pricing/calculator/`

6.2.3.5. Effect of Attacks on TEEs

An adversary model that allows a watchtower provider to break the TEE and to collude with the channel counterparty implies the risk of complete loss of a channel's funds: The channel counterparty could publish an outdated commitment transaction and the watchtower provider could extract the revocation secrets from the TEE to create a revocation transaction that directly spends the counterparty's funds in the outdated commitment transaction. In the worst case, there exists an outdated commitment transaction in which the counterparty owned the complete balance in the channel. The TEE Guard approach presents a trade-off between security and cost. We assume that there is a certain cost associated with breaking the TEE. In the extreme case, it might be that it is known that a TEE is insecure and an exploit is publicly available. If such a scenario becomes reality, customers can close their channels to protect them from malicious watchtower providers and create new channels for which only watchtower providers are hired that use TEE technologies that are considered secure.

However, even if no practical attack paths on a TEE are known, it might be that a watchtower provider can successfully attack a TEE with sufficient investment. While we assume that the cost for such an attack are high, an attack can be economical if a channel with a very large capacity is attacked. Therefore, we do not recommend the TEE Guard approach for channels with a very high capacity. Instead, users operating payment channels with a very high capacity might not need a watchtower because they are willing to pay the cost for operating a redundant infrastructure for watching the blockchain themselves.

The assessment has a different outcome if we assume that a watchtower provider is hired by many users to protect their channels that have a lower capacity and involve many different parties. While the sum of the capacities of the channels for which a watchtower provider is responsible might be large, the watchtower provider would need to collude with many different parties to steal the balances in these channels. Thus, the risk for each individual customer to outsource their channel to the watchtower provider is low. For users who have channels of low capacity, the cost for a watchtower provider are an important factor. Because TEE Guard has a low operating cost for a watchtower provider, the TEE Guard approach allows watchtower providers to charge low prices to customers.

6.2.4. Conclusion

We presented the TEE Guard watchtower approach that uses two distinctive mechanisms: Watchtower providers in TEE Guard form a distributed network to provide a higher reliability and to mutually monitor their availability. TEE Guard stores revocation secrets inside a TEE which protects them from the provider and ensures that the secrets are only used to create revocation transactions that send a channel's funds to the customer. Because revocation secrets can be stored efficiently on space logarithmic in the number of secrets, TEE Guard can be run and watch channels for low cost. In contrast to other watchtower approaches that rely on sharing revocation transactions with the watchtower, TEE Guard does also protect the amounts of HTLCs without introducing additional complexity. While

TEE Guard requires trust that the TEE is not broken, we have discussed that this risk for this scenario is low when a watchtower provider watches many low-capacity channels of different users. Thus, TEE Guard is an approach for deployable and secure watchtowers for Lightning.

6.3. Watchtower Approach using IPFS

In the TEE Guard watchtower approach, a customer employs one or multiple watchtowers and relies on them to protect the customer's payment channels from fraud. In the approach that we present in this section, we decouple the strong binding between a customer and a designated watchtower by eliminating the requirement of an explicitly employed watchtower. Instead, a customer makes the revocation transactions publicly accessible through IPFS. Anyone can act as a watchtower by monitoring the blockchain for outdated commitment transactions and, in case an outdated commitment transaction is found, by downloading and publishing a revocation transaction. To incentivize such decentralized monitoring, revocation transactions may embed a reward for successful enforcement, effectively establishing a 'blockchain neighborhood watch'.

This section is based upon our previous publication [BG22] with Hannes Bönisch. We introduce our design goals in Section 6.3.1 and present the architecture and the operation of the approach in Section 6.3.2. We evaluate the approach in Section 6.3.3 and conclude in Section 6.3.4.

6.3.1. Design Goals

In contrast to previous watchtower approaches, the IPFS-based watchtower approach should not depend on a customer's upfront arrangement with a watchtower. Instead, any watchtower should be able to work as a user's watchtower even if the user and the watchtower never had direct contact. Therefore, the revocation transactions should be publicly findable and accessible. To protect the privacy of the parties in a payment channel, parties who are not part of the channel should not learn intermediate balances. The approach should be deployable at reasonable cost for the customer so that using the approach is economical even for payment channels that do not have a high capacity. Watchtowers should be economically incentivized and have a low cost of running so that users can expect that there will be watchtowers monitoring the blockchain.

6.3.2. Architecture and Operation

The main idea of the IPFS-based watchtower approach is to separate the storage of a watchtower from the actual monitoring of the blockchain and to distribute both tasks to different entities. Figure 6.2 shows a high-level overview of the approach. A user who wants a payment channel to be monitored, creates a directory that contains a file for each

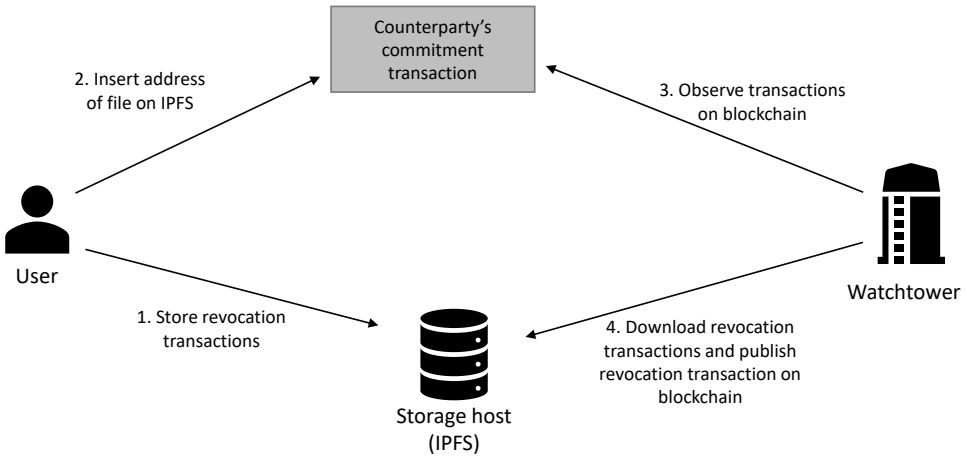

Figure 6.2.: Overview of the approach for watchtowers using IPFS. A user stores revocation transactions on a storage host. This file's address is included in each commitment transaction. If a commitment transaction is published on the blockchain, any watchtower can extract the file's address from a commitment transaction, download the revocation transactions, and publish the revocation transaction that matches the published commitment transaction if the published commitment transaction was outdated.

revocation transaction for that channel. This directory is stored on one or multiple storage hosts that make the directory accessible via IPFS. To make the address of the directory visible to a watchtower who observes a commitment transaction and wants to check if there is a matching revocation transaction, the user includes an address of the directory in every commitment transaction. Multiple independent watchtowers can monitor the blockchain for commitment transactions that include an address of an IPFS directory. If a watchtower observes such a commitment transaction, the watchtower extracts the address from the commitment transaction and accesses the directory through IPFS. If the directory contains a file for the observed commitment transaction, the watchtower downloads the file from one of the storage hosts via IPFS and extracts and publishes the revocation transaction in the Bitcoin P2P network.

Incentivization To incentivize the watchtower to monitor the blockchain and publish revocation transactions, each revocation transaction contains an output that pays a reward for a watchtower. Because the user who creates the revocation transactions does not know the watchtower, the reward is sent to an address that is generated by the user and the private key to access funds on this address is added to the file stored on IPFS. Thus, when publishing the revocation transaction, the watchtower can publish a further transaction to claim the watchtower's reward. To incentivize the storage hosts, we assume that the user pays for the storage. The storage hosts can be providers that specialize on storage hosting but are agnostic of the watchtower application. Therefore, there is a range of currently existing providers that can be used and are free to use or offer reasonable prices (see Section 6.3.3.3).

6.3.2.1. Channel Justice Directory

The information required by a watchtower to respond to cheating attempts in a payment channel is stored on IPFS in a directory called the Channel Justice Directory. The Channel Justice Directory consist of a directory file that references all files contained in the directory. For every outdated commitment transaction – i.e., all commitment transactions but the most recent commitment transaction of the channel – there is a single file that stores the information to revoke the respective commitment transaction. Each file for a commitment transaction has the hash of the commitment transaction id as name and the file contains an encrypted revocation transaction as well as an encrypted reward key. The encryption key is calculated as a cryptographic hash of the commitment transaction id concatenated with itself. This ensures that only when the commitment transaction id becomes public, the file can be decrypted. The revocation transactions are encrypted because they might leak information about payments in the channel if the associated commitment transaction includes an HTLC. The reward key is the private key for spending the reward output in the revocation transaction. It can be used by a watchtower to claim the watchtower's reward.

6.3.2.2. Operation: Payment Channel User

For the workflow of the operation of a payment channel user see Fig. 6.3. To enable publishing and updating files in IPFS using a persistent identifier, an IPNS (InterPlanetary Name System, see Section 6.1.3) address is required. When opening a new payment channel, a new IPNS key pair is generated from which the corresponding IPNS address can be deduced. The IPNS address is embedded in every commitment transaction that is created by the counterparty in the payment channel. The embedding is achieved using the OP_RETURN instruction which allows arbitrary data to be stored in a separate transaction output. To facilitate identification of transactions containing an IPNS address for Channel Justice Directory, the prefix 'CJD:' is prepended to the IPNS address. Although the IPNS address is included in every commitment transaction, it will appear at most once on the blockchain because at most one commitment transaction is ultimately published.

For each new commitment transaction that is created in the payment channel, a new file in the Channel Justice Directory is created. During an update of the payment channel, the new revocation transaction is constructed using the revocation secret obtained from the counterparty. Compared with a regular revocation transaction in Lighting, the revocation transaction in the IPFS-based watchtower approach has an additional output that rewards the watchtower. The revocation transaction as well as the private key associated with the Bitcoin address receiving the reward output are encrypted and stored inside a new file. This new file is added to IPFS and included in the Channel Justice Directory by updating the directory file of the Channel Justice Directory. Because this update changes the CID of the Channel Justice Directory, the IPNS record needs to be updated so that the IPNS address points to the newest version of the Channel Justice Directory. Once the IPNS

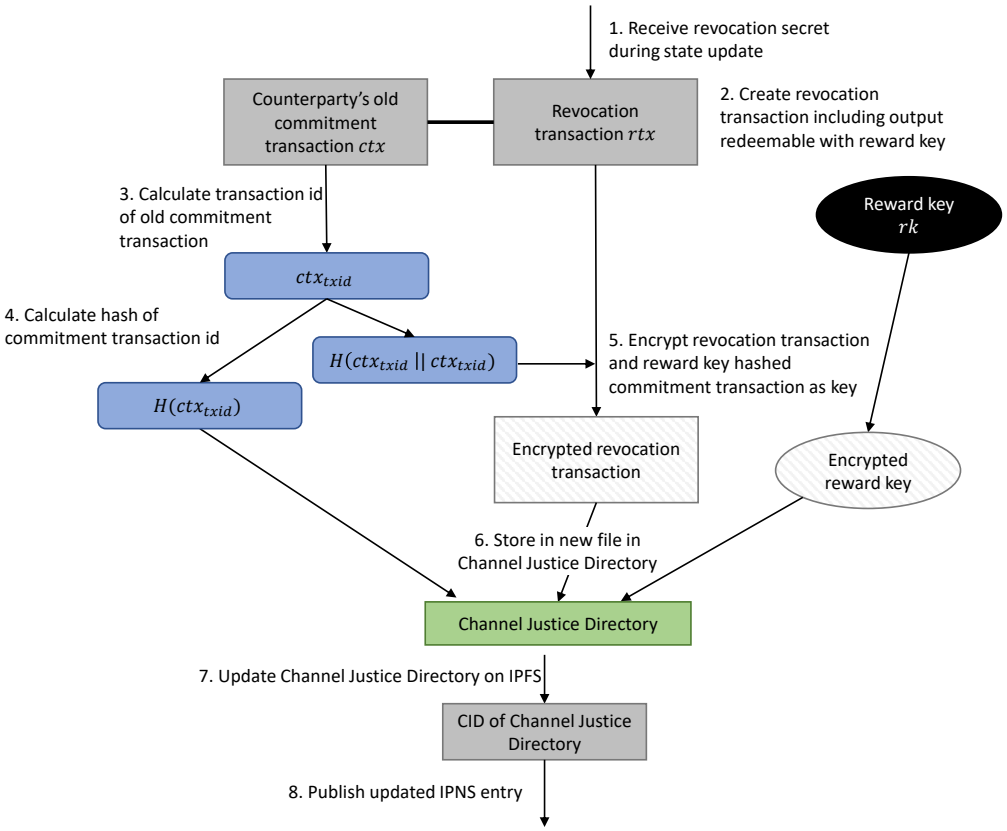

Figure 6.3.: Workflow of a user during the update of the payment channel. The user constructs the revocation transaction and generates a reward key. Both elements are encrypted and stored in a file that is added to the Channel Justice Directory. Figure adapted from [BG22].

address is updated, watchtowers can find the Channel Justice Directory via the IPNS address if a commitment transaction is published that contains the IPNS address.

6.3.2.3. Operation: Watchtower

The workflow of the operation of a watchtower is shown in Fig. 6.4. The primary function of a watchtower is to monitor the blockchain for outdated commitment transactions. When a transaction containing an IPNS address referencing the Channel Justice Directory is detected, the watchtower resolves the IPNS address and downloads the directory file of the Channel Justice Directory. If the directory contains a reference to a file named by the hash of the commitment transaction's id, the corresponding file is downloaded from a storage host found over IPFS. After decrypting the file, the watchtower publishes the revocation transaction and uses the reward key to create and publish a transaction to spend the reward output.

If no file for the commitment transaction is found in the Channel Justice Directory, this may indicate that the IPNS address has not yet been updated or that the update is still being propagated. In such cases, the watchtower periodically attempts to resolve the IPNS

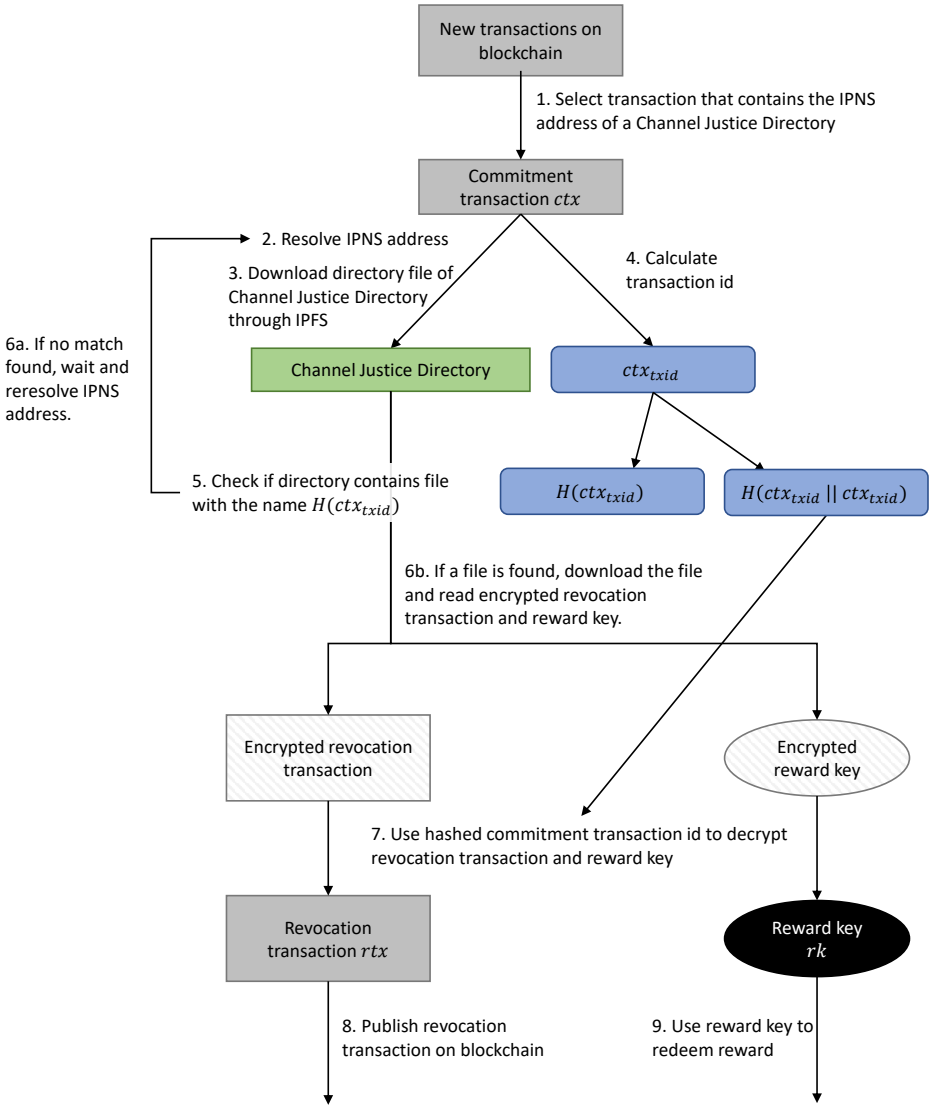

Figure 6.4.: Workflow of a watchtower. A watchtower continuously monitors the blockchain for new transactions and processes commitment transactions that include the IPNS address of a Channel Justice Directory. If the detected commitment transaction is outdated, the watchtower retrieves the corresponding file from the Channel Justice Directory, decrypts the revocation transaction, and publishes it on the blockchain. The watchtower is compensated through the reward output included in the revocation transaction. Figure adapted from [BG22].

address and re-checks for a file matching the observed commitment transaction. If no such file is found within a certain time interval, the watchtower discards the IPNS address because this outcome suggests that the observed commitment transaction is likely the most recent commitment transaction and, therefore, no revocation is required.

6.3.3. Evaluation

In the following, we analyze the IPFS-based watchtower approach with regard to reliability, scalability, deployability, and confidentiality.

6.3.3.1. Reliability

Watchtower reliability is predominantly influenced by the availability of the Channel Justice Directory. Ideally, its IPNS address remains undisclosed to the public. Nevertheless, both channel participants know the Channel Justice Directory's address and a malicious party could attempt to exhaust the bandwidth of hosts pinning the Channel Justice Directory or employ other techniques to make the file unavailable, e.g., by performing eclipse attacks on the storage hosts [PMZ21]. Such attacks can be mitigated by deploying multiple high-bandwidth servers or by leveraging DoS protection services.

Since payment channel users do not directly interact with watchtowers, they cannot engage in enforceable contracts or retrieve guarantees. Users must trust that watchtowers are operated by others. This could be achieved by trusted entities within the payment channel community running watchtowers. The incentive mechanism of the decentralized watchtower approach, the reward output, aims to encourage watchtower operation. However, rewards are contingent on frauds happening and can be claimed only by a single watchtower even when multiple watchtowers are active.

6.3.3.2. Scalability

The size of the Channel Justice Directory for a payment channel increases with each update of the payment channel. For an average update frequency of once a minute, the size of 450 byte per revocation transaction and 32 byte per reward key, the size of the raw data stored in the Channel Justice Directory accumulates to about 250 MB per year. Even with the overhead for storing the file in IPFS (e.g., for maintaining the directory file), the size of the file is practical to be stored on a storage provider. In case a commitment transaction is published, a watchtower has to download only the directory file and, if the commitment transaction is outdated, the revocation transaction for the outdated commitment transaction. If the directory file would grow too large, the directory structure could be changed to be nested to reduce the network traffic created by watchtowers.

6.3.3.3. Deployability

Operating a watchtower requires access to the Bitcoin blockchain which is typically achieved by running a Bitcoin peer. Additionally, the watchtower must retrieve files from the IPFS network which can be done either via an IPFS gateway or by running an IPFS node. Since the watchtower's primary task is to monitor new blockchain transactions for IPNS addresses, the computational requirements for the watchtower are minimal. In

the event of fraud, downloading and parsing the files from the Channel Justice Directory incurs additional bandwidth and processing overhead. Such events, however, are expected to occur rarely. Consequently, watchtower operation is inexpensive and may even be profitable if rewards are gained occasionally. However, these rewards are expected to be rather low as they depend on frauds occurring. Further, if there are multiple watchtowers, only the first one to claim the reward can be successful. A watchtower can increase the transaction fees for the transaction that claims the reward to improve the watchtower's success probability but this reduces the watchtower's profitability.

For users, costs primarily relate to off-site pinning of the Channel Justice Directory. We have estimated above, that the Channel Justice Directory for a payment channel that is updated once per minute contains 250 MB raw data per year. Assuming that the overhead for storing the data in the Channel Justice Directory is 50 % to maintain the directory index that references each file in the directory, about 375 MB need to be stored per year. Pinning this amount of data is, as of March 2026, free on Pinata and Filebase and costs a one-time fee of 5 USD on nft.storage. These low prices allow users to use multiple pinning services and get redundant hosting of the Channel Justice Directory for low cost.

6.3.3.4. Confidentiality

Although the Channel Justice Directory is publicly accessible via IPFS, it cannot be associated with a payment channel until the channel is closed through a commitment transaction. Once a commitment transaction is published, the IPNS address embedded in that transaction enables linking the Channel Justice Directory to the corresponding payment channel. Because the contents of the files in the Channel Justice Directory are encrypted, the Channel Justice Directory exposes only the number of transactions within the channel.

6.3.4. Conclusion

By removing the need for an explicitly employed watchtower, the IPFS-based watchtower approach has a low cost of using because only an IPNS address needs to be included in commitment transactions. This could even be done simply as a precaution and files could be stored on a storage host only when user goes offline and decides to actually use the IPFS-based watchtower approach. Because the watchtower operation is very simple and revocations are typically rare [Grö+26], a single watchtower could protect a whole payment channel network. However, as discussed in Section 6.1.1, the approach of storing pre-signed revocation transactions makes it difficult to cover the funds that are part of HTLCs. While this can be worked around by accepting a larger amount of stored revocation transactions and a higher complexity for both the user and a watchtower, we deem the IPFS-based watchtower approach mostly suitable for payment channels that handle only low-valued HTLCs.

6.4. Application of Payment Channel Networks: Decentralized Ticketing

In the last section of this chapter, we explore how payment channel networks can be used in an application and what benefits they offer. We do this by examining the use case of a decentralized ticketing system for public transport. We design a protocol which allows customers to buy tickets from public transport providers. By using a payment channel network, the protocol enables two crucial properties of the decentralized ticketing system: First, customers need to register only once but can buy tickets from any transportation provider in the system. Second, a customer receives a valid ticket if and only if the customer has paid for the ticket. Because these properties are inherited from the payment channel network that is used in the protocol for decentralized ticketing, this example shows that using a payment channel network as a component in protocol design can be beneficial for designing application protocols, in particular, in use cases in which customers interact with multiple different providers.

In Section 6.4.1, we present the scenario of the application and the problem statement for the ticketing protocol. Because payment channel networks do not directly offer the functionality needed by the protocol, we introduce in Section 6.4.2 adaptor signatures, PTLCs (point timelock contracts), and cryptographic commitments as additional building blocks. We present and explain the protocol in Section 6.4.3 and evaluate the protocol in Section 6.4.4. In Section 6.4.5, we discuss design options as well as benefits and limitations of the use of payment channel networks in the exemplary use case of decentralized ticketing and find that, besides the benefits that payment channel networks offer, there are still challenges left open. For example, providers as well as customers must lock liquidity in payment channels and, depending on the specific use, payment channels might become imbalanced which causes the need to rebalance channels and, thereby, increases the management complexity. We conclude in Section 6.4.6.

6.4.1. Scenario and Problem Statement

The public transport scenario includes the following actors:

- *Customers* travel using public transport.
- *Providers* operate public transport vehicles and allow customers to travel in exchange for a fare.
- Each customer is registered at a *host provider* who collects payments from the customer.
- For a specific trip of a customer, the *selling provider* is the provider who operates the vehicles on the route.

- A *query service* is an external service that provides information about routes and fares. This can, for example, be an open data platform [OCZ15; Par+18], a service offered by one of the providers, or a local database on the customers' devices.

The main use case that we focus on is that a customer wants to travel using the vehicles of any provider. For simplicity, we assume that each trip uses only the vehicles of a single provider. We refer to this provider as the selling provider. When planing the trip, the customer queries the query service for available routes and chooses a specific route that the customer wants to take. The customer buys a ticket for this specific route by paying the fee for the trip to the customer's host provider and receiving a valid ticket in return. The customer makes the trip and, during the trip, the customer can present the ticket as a proof of payment.

The idea behind integrating multiple providers into one system and having a single host provider for each customer is to improve usability for customers because a customer has to be registered only at one host provider but can buy tickets for trips offered by any provider without a second registration. A challenge in this approach is that one provider receives money for the services provided by another provider. To settle possible imbalances, the revenues of the providers need to be cleared. Previous work [Rin86; TA06; Hua+10; Yic20] has studied the problem of fare clearing: These proposals require accurate data on customer travel behavior to derive equitable allocations and assume a centralized clearing authority to manage redistribution of ticket revenues. The clearing authority computes per-provider balances and issues settlement instructions. Because the central clearing authority can levy fees, determine which providers are accepted, and observe all clearing payments, this authority must be trusted. Moreover, obtaining data on customer travel behavior that is both complete and accurate remains challenging and current collection methods yield only approximations [GW11; ZSX15; Xia+17; CZX19; Tuv+22].

By using a payment channel network for payments in such a public transport scenario, we can solve the fare clearing problem by avoiding it in the first place. We extend the public transport scenario with a payment channel network in which all providers are connected with each other either directly or indirectly. If a customer registers at the customer's host provider, the customer opens a payment channel to that provider. Using the payment channel network, this single payment channel is sufficient to send payments to any other provider. Thereby, each ticket can be paid directly to the selling provider which obviates the need for a later clearing. In this subsection, we present a protocol for ticketing based on such a scenario with a payment channel network. We start by defining the threat model and continue by introducing the properties that the protocol should achieve.

Threat Model We assume that customers and providers typically cooperate to sell and buy tickets. However, customers and providers may be malicious and we allow Byzantine behavior for customers and providers. However, we restrict malicious behavior to the digital ticket system and assume that rights granted by tickets can be enforced in the physical world. For example, we assume that customers with valid tickets are not prohibited from boarding a vehicle while customers without a valid ticket can be denied boarding

Table 6.3.: Overview of desired properties for ticket purchase and fare allocation.

Property	Description
Redistribution	The customer pays the fare to the host provider; the payment is forwarded so that the selling provider ultimately receives the amount.
Ticket Commitment	The ticket must be purchased prior to travel and be bound to a specific route and a defined validity period.
Ticket Non-Transferability	A ticket is tied to a single customer identity and cannot be transferred to another customer.
Payment Proof	The ticket serves as verifiable proof that the customer paid the ticket price to the selling provider.
Ticket Non-Duplicability	A ticket cannot be duplicated; one payment yields exactly one valid ticket.
Ticket Payment Atomicity	Ticket issuance and payment occur atomically: the customer receives a ticket if and only if the selling provider receives the fare.
Offline Ticket Inspection	Ticket validity can be verified locally by an inspector or gate without contacting any backend system.
C-SP Privacy	The selling provider learns no identifying customer information except when the selling provider is also the host provider or during inspection.
C-HP Privacy	The host provider knows only the purchase time and price; it does not learn the route, validity period, or the selling provider's identity.
C-UP Privacy	Providers not involved in a ticket purchase learn neither the ticket details nor the customer's identity.

or can be subject to enforcement actions such as fines. We assume that the query service provides correct information about routes and fares.

Problem Statement Under the specified threat model, the ticketing system should satisfy the following properties for ticket purchase and fare allocation. These properties are summarized in Table 6.3. The customer pays the fare to their host provider and the payment is redistributed within the provider network such that the selling provider ultimately receives the amount (Redistribution). A ticket must be purchased prior to travel and be bound to a specific route and validity interval (Ticket Commitment). Tickets must be non-transferable across customers and therefore be linked to a single customer identity (Ticket Non-Transferability). Each ticket must serve as cryptographic proof that the customer paid the price to the selling provider (Payment Proof). Tickets must be non-duplicable, i.e. one payment yields exactly one ticket (Ticket Non-Duplicability). To prevent accidental or deliberate failures where either a customer obtains a ticket without

paying or the selling provider receives payment without issuing a ticket, the exchange of a ticket and the payment for the ticket must be atomic: the customer receives a valid ticket if and only if the selling provider receives the ticket price (Ticket Payment Atomicity). During travel, an inspector employed by the selling provider may verify the existence and validity of a customer's ticket. This inspection to validate tickets must be possible offline, i.e., it should require only local communication between the customer and the inspector but no backend interaction (Offline Ticket Inspection). Instead of a human inspector, the ticket inspection could also be performed by a security gate at a station.

To protect customer privacy, the selling provider must not learn any identifying information about the customer except if the selling provider is also the customer's host provider or during ticket inspection (Customer-Service Provider Privacy, C-SP Privacy). Although the host provider knows the customer's identity, the host provider should learn only the purchase time and price but not the route, validity period, or the selling provider's identity (Customer-Host Provider Privacy, C-HP Privacy). Any unrelated provider must learn neither ticket details nor the customer's identity (Customer-Unrelated Provider Privacy, C-UP Privacy). While previous work [Gud15; Hin15; Han+21] addresses ticketing with stronger privacy guarantees, our problem statement encompasses both ticket sale *and* revenue allocation and our approach based on a payment channel network is decentralized.

6.4.2. Protocol Idea and Building Blocks

The Ticket Payment Atomicity property requires a fair exchange of a valid ticket in exchange for the payment for a ticket between a customer and a selling provider. A ticket is valid if and only if it is signed by the selling provider (see Section 6.4.3). Thus, a fair exchange of a signature in exchange for a payment is required. As explained in Section 2.3.6, payment channels provide a fair exchange of a payment in exchange for a secret value. To implement a fair exchange of a signature in exchange for a payment, we use adaptor signatures to complement payment channels. Adaptor signatures provide a mechanism that allows one party to create a valid signature if and only if the party knows a certain secret value.

The core idea of the ticketing protocol is the following approach: The customer receives a ticket by creating a valid signature. For creating the signature, the customer needs to know a secret value that is known by the selling provider. The customer can receive the secret value by paying the ticket's price to the selling provider. For this exchange of a ticket payment in exchange for the secret value, a payment channel network is used. Because the customer can use the secret value that is received in exchange for the payment to create the valid signature, the approach exchanges a payment for a signature.

The approach that we have just described needs a mechanism that allows for creating a signature if and only if a secret value is known. This mechanism is implemented by adaptor signatures which we will introduce in Section 6.4.2.1. The approach also needs a way to exchange a payment for a secret value. HTLCs allow for exactly such an exchange by exchanging a payment for the preimage of a given hash. Using HTLCs, the customer

receives the hash for the secret value and must send a payment to the selling provider to receive the secret value. The customer should be able to verify that the secret value that the customer is paying for is actually the value that can be used to create a valid signature. For the currently existing constructions of adaptor signatures, such a verification is not directly possible if the customer knows only the hash value of the secret value.[7] Therefore, we introduce PTLCs (point timelock contracts) in Section 6.4.2.2 which are an alternative to HTLCs in which the payment condition is the exponential of the secret value in contrast to HTLCs where the condition is the hash of the secret value. A customer can verify that the payment condition of a PTLC payment corresponds to the value required for creating a valid signature and, thus, can be assured to receive a valid signature if the customer sends the payment.

As a further building block, we introduce cryptographic commitments in Section 6.4.2.3. In the ticketing protocol that we present below, commitments are used to bind a ticket to a specific route and validity period while hiding the route and validity period from the selling provider during payment. If the customer's ticket is validated by an inspector, the customer opens the commitment and the inspector verifies that the customer's actual route matches the ticket.

In the following, we introduce adaptor signatures as well as PTLCs and cryptographic commitments as building blocks for the transport ticketing protocol. We will present the protocol in Section 6.4.3.

6.4.2.1. Adaptor Signatures

Intuitively, an adaptor signature can be understood as a locked signature that becomes only valid after unlocking: A signer creates a pre-signature for some message m. The pre-signature is locked using some key k so that it can be verified that the pre-signature is a valid pre-signature for the message m but the pre-signature is not a valid signature. Using the key k, the pre-signature can be unlocked which results in a valid signature for message m. After a valid signature has been created, the key k can be extracted by parties who know the pre-signature and the valid signature. In the context of adaptor signatures, the lock is referred to as a statement and the corresponding key is referred to as the witness. The process of unlocking is called adapting.

Adaptor signatures are useful for exchanging a signature in exchange for a witness: One party, say Alice, knows a witness y for a public statement Y and wants the other party's, say Bob's, signature on some message m. Bob wants to learn the witness y and is willing to sign the message m. Bob creates a pre-signature for the message m locking it with the public statement Y and sends the pre-signature to Alice. Alice can verify that the pre-signature is a valid pre-signature on the message m. Alice can use the witness y to adapt the pre-signature to a valid signature thereby receiving a valid signature on the

[7] We discuss in Section 6.4.5, how a customer could verify the correctness of the secret value using HTLCs for payment.

message m. From the signature, Bob can extract the witness y and, thereby, receive the witness. This approach can be used to implement a fair exchange if Alice's use of the signature can be observed by Bob. This is the case, for example, if the message m is a transaction that is published on a public blockchain.

We will use adaptor signatures in a slightly different way because we want to achieve a fair exchange of a signature in exchange for a payment. To implement this fair exchange, a selling provider creates a pre-signature for a ticket and sells the witness to adapt the pre-signature to the customer. Using a payment channel network, the customer can exchange the witness in exchange for a payment. Using the witness, the customer can obtain a signature on the ticket and, thus, obtains a valid ticket.

An adaptor signature extends a standard digital signature scheme with algorithms that produce a verifiable pre-signature that is tied to a statement/witness pair (Y, y). The statement/witness pair must be in a hard relation which means that, given a statement Y, it is computationally infeasible to find a matching witness y. Anyone who knows the corresponding witness y can adapt the pre-signature into a full signature. Formally, an adaptor signature adds the following algorithms to a standard digital signature scheme (based on [Aum+21a]):

- $\text{pSign}(m, Y, sk) \rightarrow \tilde{\sigma}$: Create a pre-signature $\tilde{\sigma}$ for message m, secret key sk, and statement Y.
- $\text{pVerify}(\tilde{\sigma}, m, Y, pk) \rightarrow \{0, 1\}$: Verify that $\tilde{\sigma}$ is a valid pre-signature for message m, statement Y, and public key pk.
- $\text{Adapt}(\tilde{\sigma}, y) \rightarrow \sigma$: Adapt the pre-signature $\tilde{\sigma}$ to a full signature σ using the witness y.
- $\text{Extract}(\tilde{\sigma}, \sigma, Y) \rightarrow y$: Recover the witness y from the pre-/full-signature pair $(\tilde{\sigma}, \sigma)$ and the statement Y.

In a secure adaptor signature scheme, it must hold that

- seeing a pre-signature $\tilde{\sigma}$ and the statement Y does not allow an adversary to forge a full-signature σ on the message m without knowing the witness y (*Existential Unforgeability*) and
- valid pre-signatures are adaptable with the correct witness (*Pre-signature Adaptability*).

There exist constructions for adaptor signatures based on ECDSA and Schnorr signatures that are proven secure [Aum+21a; Erw+21]. One application of adaptor signatures is the implementation of PTLCs that are introduced in the following.

6.4.2.2. PTLCs

Point timelock contracts (PTLCs) are an alternative to HTLCs (hashed timelock contracts). While HTLCs include a hash value in the contract's condition and can be fulfilled using a preimage for the given hash, PTLCs include an elliptic curve public key in the contract's condition (which is a point on an elliptic curve) and can be fulfilled using the corresponding private key. In elliptic curve cryptography, a private key is a scalar k and the corresponding public key is $K = kG$ where G is the base point of a given elliptic curve. Bitcoin uses the elliptic curve secp256k1 whose parameters including the base point are standardized in the Standards for Efficient Cryptography 2 (SEC 2) [Bro10]. Each private key k and corresponding public key $K = kG$ are elements of a hard relation based on the elliptic curve discrete logarithm problem (ECDLP).

On a high level, PTLCs can be used for multi-hop payments in a payment channel network in the same way that HTLCs are used: The receiver of a payment draws a random private key k, calculates the corresponding public key $K = kG$, and sends K to the sender of the payment. Beginning at the payment's sender, PTLCs using the public key K as condition are added to the payment channels on a route through the payment channel network. Once all PTLCs are added, the receiver of the payment resolves the incoming PTLC by sending the private k to the last hop. The private k is forwarded along the payment's route until all PTLCs up to the payment's sender have been fulfilled.

Technically, the encoding of a PTLC in a commitment transaction is fundamentally different than the encoding of an HTLC. As explained in Chapter 2, an HTLC between two parties (sender and receiver) is implemented in a Bitcoin transaction using an output with two spending conditions: The 'success' spending condition allows the receiver to spend the output providing the preimage for the given hash and the 'timeout' spending condition allows the sender to spend the output after the given timelock. A PTLC is implemented also using an output with two spending conditions[8]: The 'timeout' spending condition allows the sender to spend the output after the given timelock which is the same condition as for an HTLC. The 'success' spending condition requires signatures by both the sender and the receiver. Both parties create a PTLC success transaction that spends the PTLC output and has an output that is spendable by the receiver. The sender pre-signs the PTLC success transaction with an adaptor signature and the statement K and sends the pre-signature to the receiver. Using the witness k, the receiver can adapt the pre-signature to a full signature for the PTLC success transaction, add the receiver's own signature, and fulfill the PTLC by publishing the PTLC success transaction. If the sender observes the signed PTLC success transaction, the sender can extract the private key k from the sender's full signature. During usual operation, PTLCs would not be on-chain fulfilled but the receiver of the payment would send the private key k directly to the sender of the payment.

[8] Description is simplified based on [Blo21]

6.4.2.3. Cryptographic Commitments

A cryptographic commitment scheme is a two-party protocol between a committer and a receiver. The committer commits to a value x using a commitment function $Com(x)$ and subsequently reveals the committed value. Commitment schemes satisfy the two fundamental security properties hiding and binding:

Hiding The receiver cannot derive any information about the committed value from the commitment alone.

Binding The committer cannot open the commitment to any value other than the value that the committer originally committed to.

Several practical commitment schemes have been proposed relying on different assumptions, e.g., on the existence of collision-free hash functions [HM96].

6.4.3. Architecture and Operation

For the problem statement above, we propose a protocol that uses a payment channel network (see Section 6.4.1). The payment channel network is built by payment channels between public transport providers and each customer has a payment channel to their host provider. The providers exchange information about the topology of the payment channel network and the query service learns the topology of the payment channel network as well.

Any provider willing to accept new customers publicly announces its availability and accepts incoming channel openings. When a customer registers at a provider, the customer opens a payment channel to this provider. The funds deposited into this channel constitute the customer's balance and are used to pay for tickets.

In the following, we explain the protocol which is illustrated in Fig. 6.5.

6.4.3.1. Ticket Selection

If a customer wants to buy a ticket, the customer queries the query service for routing information and available ticket offers. We treat the internal implementation of the query service as out of scope and specify only its interface. The customer sends the time for the planned trip, as well as origin, and destination to the query service. In response, the query service returns a list of ticket offers. Each offer contains the seller's identity S, the ticket's price p, and a validity scope V specifying the route and the ticket's validity time window. In addition, the query service provides the customer with routing information for the payment channel network.

The customer selects one offer and constructs a pre-ticket that contains commitments to the customer's identity and to the validity scope. Concretely, the pre-ticket is represented as $(Com(I), Com(V), p)$ where $Com(I)$ is a commitment to the customer's identity I,

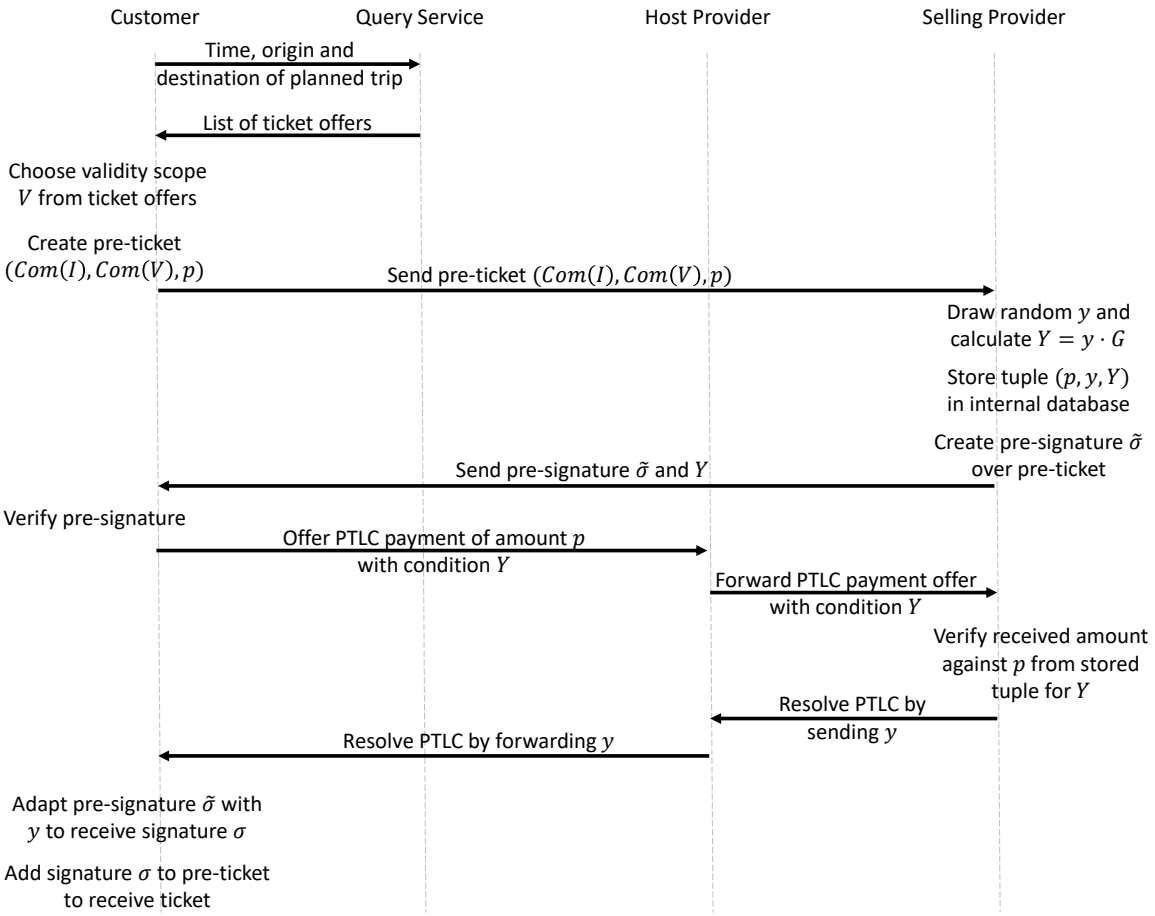

Figure 6.5.: Overview of ticket purchase protocol. The customer queries the query service for ticket offers and selects one of the offers with a specific validity scope and price. The customer sends the pre-ticket to the selling provider who creates a pre-signature on the pre-ticket. Using the payment channel network, the customer pays the ticket's price and receives the secret y. The customer adapts the pre-signature with y and receives a valid signature for the pre-ticket. The signed pre-ticket constitutes a valid ticket. Figure adapted from [GvZH22].

$Com(V)$ is a commitment to the validity scope V, and p is the ticket's price. The pre-ticket becomes a valid ticket only after it is signed by the selling provider.

6.4.3.2. Payment for Ticket

To ensure atomicity, i.e. that the customer pays for the ticket if and only if the customer receives a valid signature on the pre-ticket, we implement an fair exchange using adaptor signatures in conjunction with PTLCs. The protocol proceeds in the following steps (see Fig. 6.5).

1. The customer sends the pre-ticket $(Com(I), Com(V), p)$ to the selling provider over an encrypted channel.

2. The seller samples a random secret y and computes the statement $Y = yG$ where G is the base point of an elliptic curve.
3. The seller stores the tuple (p, y, Y) in an internal database.
4. The seller generates a pre-signature $\tilde{\sigma}$ over the pre-ticket $(Com(I), Com(V), p)$. The seller sends the pre-signature $\tilde{\sigma}$ and the statement Y to the customer. The customer can adapt the pre-signature into a valid signature once the customer learns the secret y.
5. After verifying that $\tilde{\sigma}$ is a valid pre-signature for the pre-ticket $(Com(I), Com(V), p)$, the customer computes a payment route through the payment channel network to the seller S. The customer then initiates a multi-hop payment by offering its host provider a PTLC for amount p and the statement Y as the challenge. The customer embeds onion-encrypted forwarding information for the remaining hops.
6. Upon receiving the PTLC, the seller verifies that the amount matches the expected price associated with the statement Y.
7. The seller resolves the PTLC by revealing the secret y to its neighbor who forwards the secret y back along the route. This resolves each intermediate PTLC and ultimately the secret y is sent to the customer.
8. Once the customer obtains the secret y, the customer adapts the pre-signature $\tilde{\sigma}$ into a valid signature σ on the pre-ticket $(Com(I), Com(V), p)$. The resulting signed pre-ticket $(Com(I), Com(V), p)$ together with the selling provider's public key constitutes a valid ticket.

6.4.3.3. Ticket Inspection

Ticket inspection may occur either when a customer boards a vehicle or by an inspector during the trip. An inspector who is employed by the selling provider knows the selling provider's public key. The inspector verifies the signature on the ticket and the commitments included in the ticket. During inspection, the customer transmits the ticket $(Com(I), Com(V), p)$ and the selling provider's signature σ to the inspector and opens the commitments. The communication between the inspector and the customer may be implemented, for example, via NFC or by presenting a barcode. The inspector verifies that the opened values for the customer identity I and the validity scope V are consistent with the commitments $Com(I)$ and $Com(V)$ and that σ is a valid signature on $(Com(I), Com(V), p)$ for the selling provider's public key. To enforce ticket non-transferability, the customer must reveal their identity and the inspector must confirm that the presenting individual can identify as the identity I that the ticket is committed to.

6.4.4. Evaluation

The proposed protocol satisfies the required security and privacy properties introduced above. We briefly sketch how each of the properties is achieved. The Redistribution property (i.e., the customer pays the fare to the host provider who forwards it to the selling provider) is achieved through multi-hop payments of the payment channel network by sending the payment for a ticket to the selling provider. The Ticket Commitment property (i.e., a ticket is bound to a specific route and a defined validity period) is achieved through the binding property of the commitment to the validity scope that is part of the ticket. During ticket inspection, the route and validity period are validated. Similarly, the property Ticket Non-Transferability (i.e., a ticket is bound to a customer) is achieved through the commitment to the customer's identity that is part of the ticket. The Payment Proof property (i.e., a ticket serves as a verifiable proof that the customer has paid the ticket price) is achieved through the selling provider's signature on the ticket which the customer can only obtain by adapting the pre-signature with the secret y which the customer can only obtain by paying the ticket's price to the selling provider. The property Ticket Non-Duplicability (i.e., a ticket cannot be duplicated) is achieved through the Ticket Commitment and the Ticket Non-Transferability properties because a ticket is only valid for one specific route and time and one specific customer. The Ticket Payment Atomicity property (i.e., the customer receives a ticket if and only if the selling provider receives the fare) is achieved through the atomicity of multi-hop payments based on PTLCs or HTLCs and by the Existential Unforgeability property of the adaptor signature scheme (i.e., a pre-signature cannot be adapted to a full signature without knowledge of the secret y). Offline Ticket Inspection (i.e., a ticket's validity can be verified locally) is achieved because ticket inspection only requires verification of a ticket's signature and of the commitments.

To achieve Customer-Service Provider Privacy (C-SP Privacy), we assume that the exchange of the pre-ticket and the corresponding pre-signature between the customer and the selling provider occurs over a communication channel that does not reveal the customer's identity. In practice, this exchange could be conducted via the selling provider's web interface provided that the selling provider cannot reliably identify the customer through network-level or device-level metadata (e.g., IP address, browser fingerprinting, or other side channels). Alternatively, the communication channel could be built via a privacy preserving communication system such as Tor [DMS04] or Nym [DHK21]. Because of the hiding property of the commitments to the validity scope and the customer's identity, the selling provider does not learn anything from the data contained in the pre-ticket.

Depending on the underlying fare scheme, the host provider or intermediate providers on the payment path within the payment channel network may be able to infer information about the ticket price p (e.g., a coarse estimate of trip length or fare zone). However, the protocol does not reveal the customer's precise origin or destination nor the specific intended travel time. If a specific price point uniquely corresponds to a single selling provider, observing the price p could allow an adversary to infer the seller's identity (see [Fet25, Section 3.6]). Consequently, the fare design should avoid uniquely identifying prices. Given that, Customer-Host Provider Privacy (C-HP Privacy) is achieved because

the host provider does not receive a pre-ticket or a ticket but observes only the payment from which the host provider can only deduce purchase time and price.

The Customer-Unrelated Provider Privacy (C-UP Privacy) property is achieved because unrelated providers do not receive a pre-ticket or a ticket and do not interact with the customer. While unrelated providers might participate in a multi-hop payment, they cannot relate a payment to a specific customer.

Because the selling provider accepts arbitrary pre-tickets and stores them in a database, the system exposes an attack surface for denial of service (DoS) attacks. In particular, an adversary may generate a large number of pre-ticket requests without intending to complete the payment and thereby waste the selling provider's storage and computation resources. This risk is exacerbated by the intended privacy goal: the selling provider does not learn the customer's identity during pre-ticket creation. To mitigate this threat, the selling provider can store pre-tickets only for a bounded time interval and delete the pre-tickets if the corresponding payment is not completed within this interval. In addition, the selling provider can monitor the number of pre-tickets and apply rate limiting using out-of-protocol signals, if available, such as the customer's IP address or the customer's location.

6.4.5. Discussion

In the following, we discuss design options of the ticketing approach with respect to the payment channel network and we discuss benefits and limitations of using a payment channel network.

6.4.5.1. Design Options and Protocol Extensions

A payment channel network operates as a second layer payment system anchored to an underlying layer which is typically a blockchain such as Bitcoin. However, as we have shown in Section 5.1.3, the base layer does not need to be a blockchain but may also be realized through other settlement infrastructures including banking systems which allow the participants of the payment channel network to transact in CBDCs (Central Bank Digital Currencies) rather than cryptocurrencies. This distinction is relevant for deployability as transport providers may prefer settlement in CBDCs or other government-issued currencies.

In principle, different payment channels within the same payment channel network could rely on different base layers. If different currencies occur along a single payment path then the involved parties must agree on exchange rates. To support multiple currencies, two providers could maintain multiple parallel channels between each other where each channel is backed by a different base layer or denominated in a different currency thereby expanding the set of payment options available to customers.

A more radical alternative is to eliminate the settlement layer altogether. This approach resembles credit networks[9] which do not require on-chain collateral but instead rely on bilateral trust. In such a setting, two providers agree on mutual credit limits. A payment channel is replaced by an "I owe you" (IOU) relationship that can be used for multi-hop value transfer analogously to routing in payment channel networks. The primary trade-off is that credit networks replace collateralization by trust between peers which adds a further security assumption but improves liquidity.

In integrated public transport, a common requirement is support for joint tickets spanning segments operated by multiple providers. Although a single provider may act as the seller, the corresponding revenue must be allocated among providers according to the customer's route. An approach based on a payment channel network could use multi-hop payments to support splitting a payment across multiple providers when a customer buys a ticket. Designing a complete mechanism for such multi-party settlement remains an open direction and is left for future work.

Variant using HTLCs instead of PTLCs In a payment channel network such as the current version of Lightning that supports HTLCs but not PTLCs, the protocol can be adapted by using the hash of a payment preimage as the on-path condition, while still using an adaptor signature based mechanism for a fair exchange of the ticket against a payment for the ticket – albeit with additional constraints. Concretely, the selling provider samples a secret y and computes not only the statement $Y = yG$ but also the hash value $h = H(y)$ for the HTLC condition. The selling provider then stores the tuple $(p, y, Y, H(y))$ in their database. The selling provider creates the pre-signature using the statement Y and sends the customer the pre-signature along with the hash value h. In contrast to a PTLC-based design, the customer cannot directly validate that the received pre-signature corresponds to the HTLC condition because the adaptor binding uses the statement Y while the payment condition uses the hash value h and, in general, the statement $Y = yG$ does not equal the hash value $h = H(y)$. To restore verifiability, the selling provider could additionally send the statement Y and a zero-knowledge proof that the same secret y was used to calculate the statement Y and the hash value h. However, generating and verifying such proofs can be computationally expensive and may enlarge the selling provider's attack surface for DoS, e.g., adversaries might request many proofs without completing purchases. With this caveat, atomicity still works operationally: the customer initiates an HTLC-routed payment for amount p conditioned on revealing a preimage for the hash value h. The selling provider fulfills the HTLC by releasing the secret y which propagates back to the customer. Upon receiving the secret y, the customer adapts the pre-signature into a valid signature and, thereby, completes the ticket.

[9] According to Dandekar et al. [Dan+15], the credit network model was invented independently by at least four distinct groups [DB05; Gho+07; Kar+09; Mis+08] and the term 'credit network' was introduced in 2011 by Dandekar et al. [Dan+11].

6.4.5.2. Benefits and Limitations

In the following, we focus on various aspects of the presented ticketing approach and discuss benefits and limitations of using a payment channel network.

A benefit of an approach for ticketing based on a payment channel network is that the approach removes the need for a central clearing entity. Without a central party, providers retain operational autonomy and avoid the ongoing cost and governance dependencies associated with a central clearing party. The corresponding limitation is that decentralization shifts complexity from a single operator to the ecosystem: providers must coordinate on interoperability and maintain a shared protocol which incurs additional cost for communication and agreement.

Payment channel networks offer a cost advantage because payments are executed off-chain thereby avoiding fees on the base layer for every ticket purchase and subsequent redistribution step. The limitation is that these savings depend on channel availability and liquidity: when channels must be opened, closed, or rebalanced frequently base-layer interactions reappear and can diminish the fee advantage.

Payment channel network protocols natively provide atomicity for multi-hop payments ensuring that routed transfers either complete in full or fail. In the presented approach, the same mechanism is used to achieve a fair exchange of a payment in exchange for a ticket thereby reducing the trust that a customer must place in the selling provider and vice versa.

A further benefit is finality. Once a payment is completed in the payment channel network, the payment cannot be unilateraly reversed. This reduces providers' exposure to chargebacks and lowers revenue uncertainty since paid fares are effectively settled upon payment completion. However, refunds are not natively supported. A refund can be implemented via payments in the reverse direction. To route a refund payment, however, a customer has to disclose their identity to the selling provider.

In contrast to classical post-hoc clearing, a payment channel network can realize 'implicit clearing' as a direct byproduct of the payment flow. The ticket price is paid to the host provider and forwarded to the selling provider during ticket purchase eliminating the need for later redistribution and inter-provider credit risk.

The approach based on a payment channel network can strengthen privacy by decoupling ticket issuance from payment settlement. The selling provider issues a signature on the ticket while the payment itself is routed through the payment channel network without requiring direct interaction between the customer and the selling provider. During the payment phase, information about the booked route can remain hidden because the transfer is bound only to the statement Y. However, privacy can still be affected by side information such as distinctive price points or network-level metadata. Furthermore, certain operations such as refunds may reintroduce identity disclosure.

The principal operational drawback of payment channel networks is the need to lock liquidity in channels. Customers must pre-fund a channel with their host provider before

they can purchase tickets. While the protocol protects them against loss of their funds, the locked balance represents capital that cannot be used for other purposes. Providers must commit substantial liquidity across channels to sustain routing.

Payment channels impose a liveness assumption: participants must be able to react if a counterparty attempts to close a channel in an outdated state. This requires monitoring the base layer and introduces operational complexity. Outsourcing monitoring to watchtowers can mitigate this requirement but their deployability depends on the specific use case (see Sections 6.2 and 6.3). Low-value payment use cases, such as those arising in local public transport, appear to be well suited to the application of watchtowers.

A practical limitation in real deployments is channel imbalance. If payments predominantly flow in one direction, channel capacity becomes skewed and multi-hop payments may fail because routing requires sufficient outbound liquidity at each hop. Rebalancing can restore usability but may required additional transactions (potentially on the base layer) or operational intervention which increases cost and complexity. In the use case of public transport, this applies in particular to the payment channel between a customer and the customer's host provider.

Finally, while the protocol enables redistribution of ticket fares, it does not address the allocation of public transport subsidies which often constitute a significant portion of a provider's overall revenue. To allocate subsidies, an additional mechanism would be required.

6.4.6. Conclusion

The application of ticketing for public transport shows that a payment channel network can be a useful component for the design of protocols. The architecture of a payment channel network and the mechanism of multi-hop payments enable a customer to send payments to every provider in the system although the customer is registered only at the customer's host provider. This approach generalizes to other user cases of multi provider systems in which users are registered at one provider but use services of other providers as well, e.g., electrical vehicle charging or cell phone providers. Further, the fair exchange of a ticket and the payment for the ticket is directly based on the payment channel network and is a crucial property because it reduces the required trust between customer and provider.

6.5. Conclusion

In this chapter, we have presented two watchtower approaches that aid users in handling the requirement of payment channels to regularly watch the blockchain. With the TEE Guard approach, we showed that TEEs enable scalable watchtowers that can protect their customers for small cost. Because the approach is compatible with the current Lightning protocol and because of its scalability, we deem it a good candidate for users

looking for a watchtower. With the IPFS-based watchtower approach, we showed that the storage task and the monitoring task of a watchtower can be distributed to two different parties. Thereby, the task of a watchtower is decentralized so that everyone can start a watchtower and monitor payment channels. Because a single watchtower can protect a whole payment channel network and a watchtower serves, to some extent, as deterrence, the IPFS-based watchtower approach attaches a high risk to deliberate publication of outdated commitment transactions. Finally, we showed at the application of public transport ticketing that payment channel networks are suitable for deployment in practical protocols and enable novel approaches to relevant challenges particularly in multi-provider environments.

7. Conclusions and Outlook

Payment channel networks promise low latency, low cost, and high throughput for payments by shifting most activity off-chain while retaining the underlying layer as an arbiter and trust anchor. Our work examined security aspects of payment channel networks at the example of the Lightning Network and explored the guarantees of the underlying layer, the security of the protocol itself, and security aspects of practical deployments. A central insight is that the security of a full-fledged payment channel network protocol is challenging to verify mechanically end to end. Our approach of model checking allowed us to explore and verify only small instances exhaustively. This motivates future work toward unbounded verification or a machine-checked proof of the full protocol. This step will probably be facilitated by future advances in tooling and automated proof search where recent work has shown that large language models can be effective [BLP26].

A foundational limitation is that the assumptions that payment channel networks rely on are, in theory, not strictly guaranteed by Bitcoin. In practice, however, the operational characteristics such as the typical transaction confirmation time tend to make these assumptions hold often enough so that payment channel networks function reliably. While this can be seen as sufficient from a practical point of view, future work is needed to explore how to provide stronger guarantees on the first layer or how to design payment channel network protocols to adapt weaker assumptions to maintain security given the guarantees provided by existing first layer infrastructures.

From a usability perspective, two challenges stand out. First, users must reliably monitor the first layer to react to adversarial channel closures in time. Second, capital must be locked in channels. The latter depends on the specific use case and payment channel network. If users connect with a single channel to a well-provisioned payment channel hub, users are able to send payments to other users in the payment channel network by having funded just a single channel which is operationally similar to keeping a balance with a bank. The challenge of timely monitoring can be mitigated via watchtowers. We demonstrated that TEE Guard provides a deployable approach that even protects amounts held in HTLCs, thereby improving on watchtower approaches that require sharing revocation transactions. Looking ahead, an interesting question is how to integrate watchtowers with our RFL model that restricts the visibility of transactions. In the RFL model, transactions related to a payment channel cannot be seen by third parties. Thus, to allow for watchtowers, the RFL model needs to be adapted: Either the visibility of transactions could be changed or watchtowers could be integrated into the first layer.

Beyond the current typical architecture, payment channel networks could operate in the future on top of alternative first layers. In particular, central bank digital currencies

(CBDCs) could offer a first layer for implementing payment channel networks. By providing fast and, in some aspects, cash-like digital payments, payment channel networks based on CBDCs may provide a way to enhance the deployability and acceptance of CBDCs.

We also evaluated payment channel networks as building blocks in higher-level protocols. Our case study in the context of public transport ticketing showed that payment channel networks can realize a fair exchange between a digital item (e.g., a ticket) and a payment which illustrates that payment channel networks can support other workflows beyond simple value transfer. An avenue for future work is to further explore how applications can benefit from the properties and guarantees of payment channel networks.

To summarize, we revisit our initial motivating question whether Lightning is secure and practical. Indeed, our model checking of Lightning gives high confidence that Lightning manages user balances in a secure way given that Lightning's requirements for the underlying layer are fulfilled. We have found that, from a practical point of view, Bitcoin fulfills the requirements that Lightning imposes on the underlying layer. However, Bitcoin does not provide theoretical guarantees that transactions will be executed in a timely manner, which poses a small risk to the funds held in a payment channel. We conclude that, while there are challenges left open for showing and improving the security of Lightning, the current version of Lightning can be considered secure provided that no large amounts are used that would make sophisticated attacks against the underlying layer attractive. A main use case where Lightning plays to its strengths, are micropayments, i.e. payments of low value. For use cases involving micropayments, users particularly benefit from low transaction fees of Lightning and, as it suffices to deposit small amounts in a payment channel, Lightning can be regarded as secure. Thus, for the future, Lightning might offer an interesting approach for Internet-native payments possibly used by AI agents. With respect to practicality, we conclude from our exploration of the use case of ticketing in public transport systems that Lightning can indeed be practical. The discussed use case shows that Lightning can be used as component in higher-level protocols to achieve useful properties while approaches such as watchtowers make Lightning's requirements for end users practical.

Bibliography

[AAM22] Lukas Aumayr, Kasra Abbaszadeh, and Matteo Maffei. "Thora: Atomic and Privacy-Preserving Multi-Channel Updates". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022/11/07, pp. 165–178. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560556. URL: https://dl.acm.org/doi/10.1145/3548606.3560556.

[AD90] Rajeev Alur and David Dill. "Automata for Modeling Real-Time Systems". In: *Automata, Languages and Programming*. Ed. by Michael S. Paterson. Berlin, Heidelberg: Springer, 1990, pp. 322–335. ISBN: 978-3-540-47159-2. DOI: 10.1007/BFb0032042.

[AL91] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Theoretical Computer Science* 82.2 (1991/05/31), pp. 253–284. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90224-P. URL: https://www.sciencedirect.com/science/article/pii/030439759190224P.

[And+14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. "Modeling Bitcoin Contracts by Timed Automata". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Axel Legay and Marius Bozga. Cham: Springer International Publishing, 2014, pp. 7–22. ISBN: 978-3-319-10512-3. DOI: 10.1007/978-3-319-10512-3_2.

[Aso98] N. Asokan. "Fairness in Electronic Commerce". PhD thesis. CAN: University of Waterloo, 1998/11.

[AT22] Zeta Avarikioti and Orfeas Stefanos Thyfronitis Litos. "Suborn Channels: Incentives Against Timelock Bribes". In: *Financial Cryptography and Data Security*. Ed. by Ittay Eyal and Juan Garay. Cham: Springer International Publishing, 2022, pp. 488–511. ISBN: 978-3-031-18283-9. DOI: 10.1007/978-3-031-18283-9_24.

[ATW20] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. "Cerberus Channels: Incentivizing Watchtowers for Bitcoin". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020/02, pp. 346–366. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_19.

[Aum+21a] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. "Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures". In: *Advances in Cryptology – ASIACRYPT 2021*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 635–664. ISBN: 978-3-030-92075-3. DOI: 10.1007/978-3-030-92075-3_22.

[Aum+21b] Lukas Aumayr, Matteo Maffei, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Siavash Riahi, Kristina Hostáková, and Pedro Moreno-Sanchez. "Bitcoin-Compatible Virtual Channels". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). 2021/05, pp. 901–918. DOI: 10.1109/SP40001.2021.00097.

[Aum+21c] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. "Blitz: Secure {Multi-Hop} Payments Without {Two-Phase} Commits". In: 30th USENIX Security Symposium (USENIX Security 21). 2021, pp. 4043–4060. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/aumayr.

[Aum+22] Lukas Aumayr, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. "Sleepy Channels: Bi-directional Payment Channels without Watchtowers". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022/11/07, pp. 179–192. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3559370. URL: https://dl.acm.org/doi/10.1145/3548606.3559370.

[Aum+23] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. "Breaking and Fixing Virtual Channels: Domino Attack and Donner". In: *Proceedings 2023 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society, 2023. DOI: 10.14722/ndss.2023.24370. URL: https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f370_paper.pdf.

[Aum+24] Lukas Aumayr, Esra Ceylan, Yannik Kopyciok, Matteo Maffei, Pedro Moreno-Sanchez, Iosif Salem, and Stefan Schmid. "Optimizing Virtual Payment Channel Establishment in the Face of On-Path Adversaries". In: *2024 IFIP Networking Conference (IFIP Networking)*. 2024 IFIP Networking Conference (IFIP Networking). 2024/06, pp. 1–10. DOI: 10.23919/IFIPNetworking62109.2024.10619889. URL: https://ieeexplore.ieee.org/document/10619889.

[Ava+18a] Georgia Avarikioti, Gerrit Janssen, Yuyi Wang, and Roger Wattenhofer. "Payment Network Design with Fees". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 76–84. ISBN: 978-3-030-00305-0.

[Ava+18b] Georgia Avarikioti, Felix Laufenberg, Jakub Sliwinski, Yuyi Wang, and Roger Wattenhofer. "Towards Secure and Efficient Payment Channels". In: *ArXiv181112740 Cs* (2018/11/30). arXiv: 1811.12740 [cs]. URL: http://arxiv.org/abs/1811.12740.

[Ava+20] Zeta Avarikioti, Lioba Heimbach, Yuyi Wang, and Roger Wattenhofer. "Ride the Lightning: The Game Theory of Payment Channels". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Cham: Springer International Publishing, 2020, pp. 264–283. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_15.

[Ava+21] Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. "Brick: Asynchronous Incentive-Compatible Payment Channels". In: *Financial Cryptography and Data Security*. 2021, p. 29.

[Ava+22] Zeta Avarikioti, Krzysztof Pietrzak, Iosif Salem, Stefan Schmid, Samarth Tiwari, and Michelle Yeo. "Hide & Seek: Privacy-Preserving Rebalancing on Payment Channel Networks". In: *Financial Cryptography and Data Security*. Ed. by Ittay Eyal and Juan Garay. Cham: Springer International Publishing, 2022, pp. 358–373. ISBN: 978-3-031-18283-9. DOI: 10.1007/978-3-031-18283-9_17.

[Ava+24] Zeta Avarikioti, Paweł Kędzior, Tomasz Lizurej, and Tomasz Michalak. "Bribe & Fork: Cheap PCN Bribing Attacks via Forking Threat". In: *6th Conference on Advances in Financial Technologies (AFT 2024)*. Ed. by Rainer Böhme and Lucianna Kiffer. Vol. 316. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 11:1–11:22. ISBN: 978-3-95977-345-4. DOI: 10.4230/LIPIcs.AFT.2024.11. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2024.11.

[Bad+17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. "Bitcoin as a Transaction Ledger: A Composable Treatment". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 324–356. ISBN: 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_11.

[Ban24] European Central Bank. "Study on the Payment Attitudes of Consumers in the Euro Area 2024". In: (2024/12). DOI: doi:10.2866/2856633.

[Bas+22] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. "Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols". In: *IEEE Security & Privacy* 20.3 (2022/05), pp. 24–32. ISSN: 1558-4046. DOI: 10.1109/MSEC.2022.3154689. URL: https://ieeexplore.ieee.org/document/9768326.

[BCC22] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. "ProVerif with Lemmas, Induction, Fast Subsumption, and Much More". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022 IEEE Symposium on Security and Privacy

(SP). San Francisco, CA, USA: IEEE, 2022/05, pp. 69–86. ISBN: 978-1-6654-1316-9. DOI: 10.1109/SP46214.2022.9833653. URL: https://ieeexplore.ieee.org/document/9833653/.

[BCV20] Silvia Bartolucci, Fabio Caccioli, and Pierpaolo Vivo. "A Percolation Model for the Emergence of the Bitcoin Lightning Network". In: *Scientific Reports* 10.1 (2020/03/11), p. 4488. ISSN: 2045-2322. DOI: 10.1038/s41598-020-61137-5. URL: https://www.nature.com/articles/s41598-020-61137-5.

[BDW17] Conrad Burchert, Christian Decker, and Roger Wattenhofer. "Scalable Funding of Bitcoin Micropayment Channel Networks". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Paul Spirakis and Philippas Tsigas. Vol. 10616. Cham: Springer International Publishing, 2017, pp. 361–377. ISBN: 978-3-319-69083-4 978-3-319-69084-1. DOI: 10.1007/978-3-319-69084-1_26. URL: http://link.springer.com/10.1007/978-3-319-69084-1_26.

[Ben+24] Marco Benedetti, Francesco De Sclavis, Marco Favorito, Giuseppe Galano, Sara Giammusso, Antonio Muci, and Matteo Nardelli. "Self-Balancing Semi-Hierarchical Payment Channel Networks for Central Bank Digital Currencies". In: *2024 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops)*. 2024 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops). 2024/03, pp. 530–536. DOI: 10.1109/PerComWorkshops59983.2024.10503409. URL: https://ieeexplore.ieee.org/document/10503409.

[Ben+25] Marco Benedetti, Francesco De Sclavis, Marco Favorito, Giuseppe Galano, Sara Giammusso, Antonio Muci, and Matteo Nardelli. "An Analysis of Pervasive Payment Channel Networks for Central Bank Digital Currencies". In: *Computer Communications* (2025/05/12), p. 108199. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2025.108199. URL: https://www.sciencedirect.com/science/article/pii/S0140366425001562.

[Ben14] Juan Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *ArXiv14073561 Cs* (2014/07/14). arXiv: 1407.3561 [cs]. URL: http://arxiv.org/abs/1407.3561.

[BG22] Hannes Bönisch and Matthias Grundmann. "Decentralizing Watchtowers for Payment Channels Using IPFS". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW). 2022/07, pp. 27–32. DOI: 10.1109/ICDCSW56584.2022.00015. URL: https://ieeexplore.ieee.org/document/9951162.

[BGW20] Colin Boyd, Kristian Gjøsteen, and Shuang Wu. "A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract". In: *DROPS-IDN/v2/Document/10.4230/OASIcs.FMBC.2020.5*. 2nd Workshop on Formal Methods for Blockchains (FMBC 2020). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/OASIcs.FMBC.2020.5. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2020.5.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008/04/25. 984 pp. ISBN: 978-0-262-02649-9. URL: https://mitpress.mit.edu/9780262026499/principles-of-model-checking/.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. "Deanonymisation of Clients in Bitcoin P2P Network". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014/11/03, pp. 15–29. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660379.

[Bla16] Bruno Blanchet. "Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif". In: *Foundations and Trends in Privacy and Security* 1.1–2 (2016/10/31), pp. 1–135. ISSN: 2474-1558. DOI: 10.1561/3300000004. URL: https://doi.org/10.1561/3300000004.

[Bla22] Bruno Blanchet. "The Security Protocol Verifier ProVerif and Its Horn Clause Resolution Algorithm". In: *Electronic Proceedings in Theoretical Computer Science* 373 (2022/11/22), pp. 14–22. ISSN: 2075-2180. DOI: 10.4204/EPTCS.373.2. arXiv: 2211.12227 [cs]. URL: http://arxiv.org/abs/2211.12227.

[BLP26] Massimo Bartoletti, Enrico Lipparini, and Livio Pompianu. "LLMs as Verification Oracles for Solidity". In: Financial Cryptography and Data Security. 2026. URL: https://fc26.ifca.ai/preproceedings/253.pdf.

[BNT20] Vivek Bagaria, Joachim Neu, and David Tse. "Boomerang: Redundancy Improves Latency and Throughput in Payment-Channel Networks". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Cham: Springer International Publishing, 2020, pp. 304–324. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_17.

[BNT22] Alex Biryukov, Gleb Naumenko, and Sergei Tikhomirov. "Analysis and Probing of Parallel Channels in the Lightning Network". In: *Financial Cryptography and Data Security*. Ed. by Ittay Eyal and Juan Garay. Cham: Springer International Publishing, 2022, pp. 337–357. ISBN: 978-3-031-18283-9. DOI: 10.1007/978-3-031-18283-9_16.

[Bob+11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. "Why3: Shepherd Your Herd of Provers". In: Boogie 2011: First International Workshop on Intermediate Verification Languages. 2011, p. 53. URL: https://inria.hal.science/hal-00790310.

[Bou+18] Patricia Bouyer, Uli Fahrenberg, Kim Guldstrand Larsen, Nicolas Markey, Joël Ouaknine, and James Worrell. "Model Checking Real-Time Systems". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 1001–1046. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_29. URL: https://doi.org/10.1007/978-3-319-10575-8_29.

[Boz+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. "Kronos: A Model-Checking Tool for Real-Time Systems". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by Anders P. Ravn and Hans Rischel. Berlin, Heidelberg: Springer, 1998, pp. 298–302. ISBN: 978-3-540-49792-9. DOI: 10.1007/BFb0055357.

[Bro10] Daniel R. L. Brown. *SEC 2: Recommended Elliptic Curve Domain Parameters*. 2010/01/27. URL: https://www.secg.org/sec2-v2.pdf.

[Bru+23] Lea Salome Brugger, Laura Kovács, Anja Petkovic Komel, Sophie Rain, and Michael Rawson. "CheckMate: Automated Game-Theoretic Security Reasoning". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023/11/21, pp. 1407–1421. ISBN: 979-8-4007-0050-7. DOI: 10.1145/3576915.3623183. URL: https://dl.acm.org/doi/10.1145/3576915.3623183.

[BSB21] Ferenc Béres, István András Seres, and András A. Benczúr. "A Cryptoeconomic Traffic Analysis of Bitcoin's Lightning Network". In: *Cryptoeconomic Systems* 0.1 (2021/04/07). ISSN: 2767-4207, DOI: 10.21428/58320208.d4cd697e. URL: https://cryptoeconomicsystems.pubpub.org/pub/beres-lightning-traffic/release/12.

[BST98] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. "Modeling Urgency in Timed Systems". In: *Compositionality: The Significant Difference*. Ed. by Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli. Berlin, Heidelberg: Springer, 1998, pp. 103–129. ISBN: 978-3-540-49213-9. DOI: 10.1007/3-540-49213-5_5.

[But14] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2014, pp. 2–1. URL: https://cryptorating.eu/whitepapers/Ethereum/Ethereum_white_paper.pdf.

[BXW22] Qianlan Bai, Yuedong Xu, and Xin Wang. "Understanding the Benefit of Being Patient in Payment Channel Networks". In: *IEEE Transactions on Network Science and Engineering* (2022), pp. 1–1. ISSN: 2327-4697. DOI: 10.1109/TNSE.2022.3154408.

[Cam+22] Gustavo F. Camilo, Gabriel Antonio F. Rebello, Lucas Airam C. de Souza, Maria Potop-Butucaru, Marcelo D. Amorim, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. "Topological Evolution Analysis of Payment Channels in the Lightning Network". In: *2022 IEEE Latin-American Conference on Communications (LATINCOM)*. 2022 IEEE Latin-American Conference on Communications (LATINCOM). 2022/11, pp. 1–6. DOI: 10.1109/LATINCOM56090.2022.10000445. URL: https://ieeexplore.ieee.org/document/10000445.

[Can01] R. Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. Proceedings 42nd IEEE Symposium on Foundations of Computer Science. 2001/10, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[Cao+20] Tong Cao, Jiangshan Yu, Jérémie Decouchant, Xiapu Luo, and Paulo Verissimo. “Exploring the Monero Peer-to-Peer Network”. In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 578–594. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_31.

[Cas+21] Pedro Casas, Matteo Romiti, Peter Holzer, Sami Ben Mariem, Benoit Donnet, and Bernhard Haslhofer. “Where Is the Light(Ning) in the Taproot Dawn? Unveiling the Bitcoin Lightning (IP) Network”. In: *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*. 2021 IEEE 10th International Conference on Cloud Networking (CloudNet). 2021/11, pp. 87–90. DOI: 10.1109/CloudNet53349.2021.9657121. URL: https://ieeexplore.ieee.org/document/9657121.

[Cen22] Microsoft Research–Inria Joint Centre. *TLA+ Proof System*. 2022. URL: https://proofs.tlapl.us/doc/web/content/Home.html.

[CFN88] David Chaum, Amos Fiat, and Moni Naor. “Untraceable Electronic Cash”. In: *Advances in Cryptology — CRYPTO’ 88*. Conference on the Theory and Application of Cryptography. Lecture Notes in Computer Science. Springer, New York, NY, 1988/08/21, pp. 319–327. ISBN: 978-0-387-97196-4 978-0-387-34799-8. DOI: 10.1007/0-387-34799-2_25. URL: https://link.springer.com/chapter/10.1007/0-387-34799-2_25.

[Cha83] David Chaum. “Blind Signatures for Untraceable Payments”. In: *Advances in Cryptology*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Boston, MA: Springer US, 1983, pp. 199–203. ISBN: 978-1-4757-0602-4. DOI: 10.1007/978-1-4757-0602-4_18.

[Che+22] Yanjiao Chen, Xuxian Li, Jian Zhang, and Hongliang Bi. “Multi-Party Payment Channel Network Based on Smart Contract”. In: *IEEE Transactions on Network and Service Management* 19.4 (2022/12), pp. 4847–4857. ISSN: 1932-4537. DOI: 10.1109/TNSM.2022.3162592. URL: https://ieeexplore.ieee.org/abstract/document/9745518.

[Che+25] Ruonan Chen, Yang Zhang, Dawei Li, Yizhong Liu, Jianwei Liu, Qianhong Wu, Jianying Zhou, and Willy Susilo. “Bitcoin-Compatible Privacy-Preserving Multi-Party Payment Channels Supporting Variable Amounts”. In: *IEEE Transactions on Information Forensics and Security* (2025), pp. 1–1. ISSN: 1556-6021. DOI: 10.1109/TIFS.2025.3528827. URL: https://ieeexplore.ieee.org/abstract/document/10839096.

[CHX18] Jeff Coleman, Liam Horne, and Li Xuanji. *Counterfactual: Generalized State Channels*. 2018/06/12. URL: https://liamhorne.com/counterfactual.pdf.

[Cir+25] Horatiu Cirstea, Markus A. Kuppe, Benjamin Loillier, and Stephan Merz. "Validating Traces of Distributed Programs Against TLA+ Specifications". In: *Software Engineering and Formal Methods*. Ed. by Alexandre Madeira and Alexander Knapp. Cham: Springer Nature Switzerland, 2025, pp. 126–143. ISBN: 978-3-031-77382-2. DOI: 10.1007/978-3-031-77382-2_8.

[Cro+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: *Financial Cryptography and Data Security*. Ed. by Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff. Berlin, Heidelberg: Springer, 2016, pp. 106–125. ISBN: 978-3-662-53357-4. DOI: 10.1007/978-3-662-53357-4_8.

[CVD19] Marco Conoscenti, Antonio Vetrò, and Juan Carlos De Martin. "Hubs, Rebalancing and Service Providers in the Lightning Network". In: *IEEE Access* 7 (2019), pp. 132828–132840. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2941448.

[CZX19] Gang Cheng, Shuzhi Zhao, and Shengbo Xu. "Estimation of Passenger Route Choices for Urban Rail Transit System Based on Automatic Fare Collection Mined Data". In: *Transactions of the Institute of Measurement and Control* 41.11 (2019/07/01), pp. 3092–3102. ISSN: 0142-3312. DOI: 10.1177/0142331218823855. URL: https://doi.org/10.1177/0142331218823855.

[Dan+11] Pranav Dandekar, Ashish Goel, Ramesh Govindan, and Ian Post. "Liquidity in Credit Networks: A Little Trust Goes a Long Way". In: *Proceedings of the 12th ACM Conference on Electronic Commerce*. EC '11. New York, NY, USA: Association for Computing Machinery, 2011/06/05, pp. 147–156. ISBN: 978-1-4503-0261-6. DOI: 10.1145/1993574.1993597. URL: https://dl.acm.org/doi/10.1145/1993574.1993597.

[Dan+15] Pranav Dandekar, Ashish Goel, Michael P. Wellman, and Bryce Wiedenbeck. "Strategic Formation of Credit Networks". In: *ACM Transactions on Internet Technology* 15.1 (2015/03/12), 3:1–3:41. ISSN: 1533-5399. DOI: 10.1145/2700058. URL: https://dl.acm.org/doi/10.1145/2700058.

[DB05] Dd.B. DeFigueiredo and E.T. Barr. "TrustDavis: A Non-Exploitable Online Reputation System". In: *Seventh IEEE International Conference on E-Commerce Technology (CEC'05)*. Seventh IEEE International Conference on E-Commerce Technology (CEC'05). 2005/07, pp. 274–283. DOI: 10.1109/ICECT.2005.98. URL: https://ieeexplore.ieee.org/document/1524055.

[DBG18] Varun Deshpande, Hakim Badis, and Laurent George. "BTCmap: Mapping Bitcoin Peer-to-Peer Network Topology". In: *2018 IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)*. 2018 IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN). 2018/09, pp. 1–6. DOI: 10.23919/PEMWN.2018.8548904.

[Del+19] Sergi Delgado-Segura, Surya Bakshi, Cristina Pérez-Solà, James Litton, Andrew Pachulski, Andrew Miller, and Bobby Bhattacharjee. "TxProbe: Discovering Bitcoin's Network Topology Using Orphan Transactions". In: *Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 550–566. ISBN: 978-3-030-32101-7. DOI: 10.1007/978-3-030-32101-7_32.

[DeL26] Caleb James DeLisle. *Cjdelisle/Cjdns*. 2026/03/01. URL: https://github.com/cjdelisle/cjdns.

[DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. "General State Channel Networks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018/10/15, pp. 949–966. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243856. URL: https://dl.acm.org/doi/10.1145/3243734.3243856.

[DHK21] Claudia Diaz, Harry Halpin, and Aggelos Kiayias. *The Nym Network*. https://nymtech.net/nym-whitepaper.pdf. 2021/02. URL: https://nymtech.net/nym-whitepaper.pdf.

[DK23] Stefan Dziembowski and Paweł Kędzior. "Non-Atomic Payment Splitting in Channel Networks". In: *5th Conference on Advances in Financial Technologies (AFT 2023)*. Ed. by Joseph Bonneau and S. Matthew Weinberg. Vol. 282. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 17:1–17:23. ISBN: 978-3-95977-303-4. DOI: 10.4230/LIPIcs.AFT.2023.17. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2023.17.

[DM10] Mohammad Torabi Dashti and Sjouke Mauw. "Fair Exchange". In: *Handbook of Financial Cryptography and Security*. Ed. by Burton Rosenberg. Chapman and Hall/CRC, 2010, pp. 109–132. DOI: 10.1201/9781420059823-C5. URL: https://doi.org/10.1201/9781420059823-c5.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: *Proceedings of the 13th USENIX Security Symposium*. 13th USENIX Security Symposium (USENIX Security 04). 2004. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

[Don+18] Mo Dong, Qingkai Liang, Xiaozhou Li, and Junda Liu. *Celer Network: Bring Internet Scale to Every Blockchain*. arXiv, 2018/09/28. DOI: 10.48550/arXiv.1810.00037. arXiv: 1810.00037 [cs]. URL: http://arxiv.org/abs/1810.00037.

[DPH14] Joan Antoni Donet Donet, Cristina Pérez-Solà, and Jordi Herrera-Joancomartí. "The Bitcoin P2P Network". In: *Financial Cryptography and Data Security*. Ed. by Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith. Lec-

ture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 87–102. ISBN: 978-3-662-44774-1. DOI: 10.1007/978-3-662-44774-1_7.

[DRT19] Erik Daniel, Elias Rohrer, and Florian Tschorsch. "Map-Z: Exposing the Zcash Network in Times of Transition". In: *ArXiv190709755 Cs* (2019/07/23). arXiv: 1907.09755 [cs]. URL: http://arxiv.org/abs/1907.09755.

[Dry16] Thaddeus Dryja. "Unlinkable Outsourced Channel Monitoring". Scaling Bitcoin 2016 (Milan). 2016/10/08. URL: https://scalingbitcoin.org/milan2016/presentations/D1%20-%208%20-%20Tadge%20Dryja.pdf.

[DT17] Christian Decker and Anthony Towns. *BIP 118: SIGHASH_ANYPREVOUT for Taproot Scripts*. 2017/02/28. URL: https://github.com/bitcoin/bips/blob/master/bip-0118.mediawiki.

[DT98] Conrado Daws and Stavros Tripakis. "Model Checking of Real-Time Reachability Properties Using Abstractions". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Berlin, Heidelberg: Springer, 1998, pp. 313–329. ISBN: 978-3-540-69753-4. DOI: 10.1007/BFb0054180.

[DW15] Christian Decker and Roger Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Andrzej Pelc and Alexander A. Schwarzmann. Vol. 9212. Cham: Springer International Publishing, 2015, pp. 3–18. ISBN: 978-3-319-21740-6 978-3-319-21741-3. DOI: 10.1007/978-3-319-21741-3_1. URL: http://link.springer.com/10.1007/978-3-319-21741-3_1.

[Dzi+19a] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. "Multi-Party Virtual State Channels". In: *Advances in Cryptology – EUROCRYPT 2019*. Ed. by Yuval Ishai and Vincent Rijmen. Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 625–656. ISBN: 978-3-030-17653-2.

[Dzi+19b] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. "Perun: Virtual Payment Hubs over Cryptocurrencies". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, 2019/05, pp. 106–123. ISBN: 978-1-5386-6660-9. DOI: 10.1109/SP.2019.00020. URL: https://ieeexplore.ieee.org/document/8835315/.

[Eck+20] Lisa Eckey, Sebastian Faust, Kristina Hostáková, and Stefanie Roos. *Splitting Payments Locally While Routing Interdimensionally*. 555. 2020. URL: https://eprint.iacr.org/2020/555.

[EMM19] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA:

Association for Computing Machinery, 2019/11/06, pp. 801–815. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3345666. URL: https://doi.org/10.1145/3319535.3345666.

[Eng+17] Felix Engelmann, Henning Kopp, Frank Kargl, Florian Glaser, and Christof Weinhardt. "Towards an Economic Analysis of Routing in Payment Channel Networks". In: *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*. SERIAL '17. New York, NY, USA: Association for Computing Machinery, 2017/12/11, pp. 1–6. ISBN: 978-1-4503-5173-7. DOI: 10.1145/3152824.3152826. URL: https://dl.acm.org/doi/10.1145/3152824.3152826.

[Erd+21] Enes Erdin, Mumin Cebe, Kemal Akkaya, Eyuphan Bulut, and Selcuk Uluagac. "A Scalable Private Bitcoin Payment Channel Network with Privacy Guarantees". In: *Journal of Network and Computer Applications* (2021/02/18), p. 103021. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2021.103021. URL: https://www.sciencedirect.com/science/article/pii/S1084804521000485.

[ERE20] Oğuzhan Ersoy, Stefanie Roos, and Zekeriya Erkin. "How to Profit from Payments Channels". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 284–303. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_16.

[Ers+23] Oguzhan Ersoy, Jérémie Decouchant, Satwik Prabhu Kumble, and Stefanie Roos. "SyncPCN/PSyncPCN: Payment Channel Networks without Blockchain Synchrony". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. AFT '22. New York, NY, USA: Association for Computing Machinery, 2023/07/05, pp. 16–29. ISBN: 978-1-4503-9861-9. DOI: 10.1145/3558535.3559779. URL: https://dl.acm.org/doi/10.1145/3558535.3559779.

[Erw+21] Andreas Erwig, Sebastian Faust, Kristina Hostáková, Monosij Maitra, and Siavash Riahi. *Two-Party Adaptor Signatures From Identification Schemes*. 2021/150. 2021. URL: https://eprint.iacr.org/2021/150.

[Erw+23] Andreas Erwig, Sebastian Faust, Siavash Riahi, and Tobias Stöcker. "CommiTEE : An Efficient and Secure Commit-Chain Protocol Using TEEs". In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P). 2023/07, pp. 429–448. DOI: 10.1109/EuroSP57164.2023.00033. URL: https://ieeexplore.ieee.org/abstract/document/10190526.

[Fet25] Valerie Maria Fetzer. "Towards Solving Real-World Challenges for Privacy-Preserving Systems". 2025. DOI: 10.5445/IR/1000185855. URL: https://publikationen.bibliothek.kit.edu/1000185855.

[FOA16] Muntadher Fadhil, Gareth Owenson, and Mo Adda. "A Bitcoin Model for Evaluation of Clustering to Improve Propagation Delay in Bitcoin Network". In: *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. 2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES). 2016/08, pp. 468–475. DOI: 10.1109/CSE-EUC-DCABES.2016.226.

[Fro20] Conner Fromknecht. *[Lightning-dev] CVE-2020-26896: LND Invoice Preimage Extraction*. E-mail. 2020/10/20. URL: https://diyhpl.us/~bryan/irc/bitcoin/bitcoin-dev/linuxfoundation-pipermail/lightning-dev/2020-October/002857.txt.

[FSD22] Federico Franzoni, Xavier Salleras, and Vanesa Daza. "AToM: Active Topology Monitoring for the Bitcoin Peer-to-Peer Network". In: *Peer-to-Peer Networking and Applications* 15.1 (2022/01/01), pp. 408–425. ISSN: 1936-6450. DOI: 10.1007/s12083-021-01201-7. URL: https://doi.org/10.1007/s12083-021-01201-7.

[FSL26] Grzegorz Fabiański, Rafał Stefański, and Orfeas Stefanos Thyfronitis Litos. "A Formally Verified Lightning Network". In: *Financial Cryptography and Data Security*. Ed. by Christina Garman and Pedro Moreno-Sanchez. Cham: Springer Nature Switzerland, 2026, pp. 3–20. ISBN: 978-3-032-07024-1. DOI: 10.1007/978-3-032-07024-1_1.

[GBH22] Matthias Grundmann, Max Baumstark, and Hannes Hartenstein. "On the Peer Degree Distribution of the Bitcoin P2P Network". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2022/05, pp. 1–5. DOI: 10.1109/ICBC54727.2022.9805511.

[Ge+22] Zhonghui Ge, Yi Zhang, Yu Long, and Dawu Gu. "Shaduf: Non-Cycle Payment Channel Rebalancing". In: *Proceedings 2022 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA, USA: Internet Society, 2022. ISBN: 978-1-891562-74-7. DOI: 10.14722/ndss.2022.24203. URL: https://www.ndss-symposium.org/wp-content/uploads/2022-203-paper.pdf.

[Ge+23] Zhonghui Ge, Yi Zhang, Yu Long, and Dawu Gu. "Magma: Robust and Flexible Multi-Party Payment Channel". In: *IEEE Transactions on Dependable and Secure Computing* 20.6 (2023/11), pp. 5024–5042. ISSN: 1941-0018. DOI: 10.1109/TDSC.2023.3238332. URL: https://ieeexplore.ieee.org/abstract/document/10024888.

[GH20] Matthias Grundmann and Hannes Hartenstein. "Fundamental Properties of the Layer Below a Payment Channel Network". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomarti. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 409–420. ISBN: 978-3-030-66172-4. DOI: 10.1007/978-3-030-66172-4_26.

[GH22] Matthias Grundmann and Hannes Hartenstein. "Verifying Payment Channels with TLA+". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2022/05, pp. 1–3. DOI: 10.1109/ICBC54727.2022.9805487.

[GH25] Matthias Grundmann and Hannes Hartenstein. *Model Checking the Security of the Lightning Network*. arXiv, 2025/05/21. DOI: 10.48550/arXiv.2505.15568. arXiv: 2505.15568 [cs]. URL: http://arxiv.org/abs/2505.15568.

[GH26a] Matthias Grundmann and Hannes Hartenstein. "Security of the Lightning Network: Model Checking a Stepwise Refinement with TLA+". In: *Integrated Formal Methods*. Ed. by Ferruccio Damiani and Marie Farrell. Cham: Springer Nature Switzerland, 2026, pp. 313–335. ISBN: 978-3-032-10794-7. DOI: 10.1007/978-3-032-10794-7_16.

[GH26b] Matthias Grundmann and Hannes Hartenstein. *TLA+ Specification of Lightning, Security Property, and Refinement Mappings*. Zenodo, 2026/04/30. DOI: 10.5281/zenodo.19828898. URL: https://zenodo.org/records/19828898.

[Gho+07] Arpita Ghosh, Mohammad Mahdian, Daniel M. Reeves, David M. Pennock, and Ryan Fugger. "Mechanism Design on Trust Networks". In: *Internet and Network Economics*. Ed. by Xiaotie Deng and Fan Chung Graham. Berlin, Heidelberg: Springer, 2007, pp. 257–268. ISBN: 978-3-540-77105-0. DOI: 10.1007/978-3-540-77105-0_25.

[GJ24] Jared Griffiths and Matthew Joyce. *The Reliability of Retail Payment Services*. Reserve Bank of Australia, 2024/10/17. URL: https://www.rba.gov.au/publications/bulletin/2024/oct/the-reliability-of-retail-payment-services.html.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *Advances in Cryptology - EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 281–310. ISBN: 978-3-662-46803-6. DOI: 10.1007/978-3-662-46803-6_10.

[GKL20] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. *Full Analysis of Nakamoto Consensus in Bounded-Delay Networks*. 277. 2020. URL: https://eprint.iacr.org/2020/277.

[GLH19] Matthias Grundmann, Marc Leinweber, and Hannes Hartenstein. "Banklaves: Concept for a Trustworthy Decentralized Payment Service for Bitcoin". In: *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2019/05, pp. 268–276. DOI: 10.1109/BLOC.2019.8751394.

[GM17] Matthew Green and Ian Miers. "Bolt: Anonymous Payment Channels for Decentralized Currencies". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017/10/30, pp. 473–489. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134093. URL: https://doi.org/10.1145/3133956.3134093.

[GNH19] Matthias Grundmann, Till Neudecker, and Hannes Hartenstein. "Exploiting Transaction Accumulation and Double Spends for Topology Inference in Bitcoin". In: *Financial Cryptography and Data Security*. Ed. by Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala. Vol. 10958. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2019, pp. 113–126. ISBN: 978-3-662-58820-8.

[Grö+26] Florian Grötschla, Lioba Heimbach, Severin Richner, and Roger Wattenhofer. "On the Lifecycle of a Lightning Network Payment Channel". In: *Financial Cryptography and Data Security. FC 2025 International Workshops*. Ed. by Bernhard Haslhofer, Java Xu, Friedhelm Victor, Massimo Bartoletti, Andrea Bracciali, Kanta Matsuura, Jarek Nabrzyski, Vero Estrada-Galiñanes, Claudio Tessone, Jurlind Budurushi, and Karola Marky. Cham: Springer Nature Switzerland, 2026, pp. 1–16. ISBN: 978-3-032-00492-5. DOI: 10.1007/978-3-032-00492-5_1.

[Gru+22] Matthias Grundmann, Hedwig Amberg, Max Baumstark, and Hannes Hartenstein. "Short Paper: What Peer Announcements Tell Us About the Size of the Bitcoin P2P Network". In: *Financial Cryptography and Data Security*. Ed. by Ittay Eyal and Juan Garay. Cham: Springer International Publishing, 2022, pp. 694–704. ISBN: 978-3-031-18283-9. DOI: 10.1007/978-3-031-18283-9_35.

[Gud+20] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. "SoK: Layer-Two Blockchain Protocols". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 201–226. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_12.

[Gud15] Ivan Gudymenko. "Privacy-Preserving E-ticketing Systems for Public Transport Based on RFID/NFC Technologies". TU Dresden, 2015/05/26. URL: https://d-nb.info/1073206866/34.

[Gue+19] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. "The Consensus Number of a Cryptocurrency". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*

- *PODC '19*. The 2019 ACM Symposium. Toronto ON, Canada: ACM Press, 2019, pp. 307–316. ISBN: 978-1-4503-6217-7. DOI: 10.1145/3293611.3331589. URL: http://dl.acm.org/citation.cfm?doid=3293611.3331589.

[GvZH22] Matthias Grundmann, Otto von Zastrow-Marcks, and Hannes Hartenstein. "On the Applicability of Payment Channel Networks for Allocation of Transport Ticket Revenues". In: *2022 International Conference on Computer Communications and Networks (ICCCN)*. 2022 International Conference on Computer Communications and Networks (ICCCN). 2022/07, pp. 1–7. DOI: 10.1109/ICCCN54977.2022.9868881. URL: https://ieeexplore.ieee.org/document/9868881.

[GW11] Shengguo Gao and Zhong Wu. "Modeling Passenger Flow Distribution Based on Travel Time of Urban Rail Transit". In: *Journal of Transportation Systems Engineering and Information Technology* 11.6 (2011/12/01), pp. 124–130. ISSN: 1570-6672. DOI: 10.1016/S1570-6672(10)60156-0. URL: https://www.sciencedirect.com/science/article/pii/S1570667210601560.

[Han+21] Jinguang Han, Liqun Chen, Steve Schneider, Helen Treharne, and Stephan Wesemeyer. "Privacy-Preserving Electronic Ticket Scheme with Attribute-Based Credentials". In: *IEEE Transactions on Dependable and Secure Computing* 18.4 (2021/07), pp. 1836–1849. ISSN: 1941-0018. DOI: 10.1109/TDSC.2019.2940946.

[Hei+17] Ethan Heilman, Leen AlShenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. "TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2017. DOI: 10.14722/ndss.2017.23086. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/tumblebit-untrusted-bitcoin-compatible-anonymous-payment-hub/.

[Her+19] Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, Alejandro Ranchal-Pedrosa, Cristina Pérez-Solà, and Joaquin Garcia-Alfaro. "On the Difficulty of Hiding the Balance of Lightning Network Channels". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019/07/02, pp. 602–612. ISBN: 978-1-4503-6752-3. DOI: 10.1145/3321705.3329812. URL: https://doi.org/10.1145/3321705.3329812.

[Hin15] Gesine Hinterwalder. "Privacy-Preserving Payments for Transportation Systems". University of Massachusetts Amherst, 2015/11/09. URL: https://scholarworks.umass.edu/dissertations_2/527.

[HM96] Shai Halevi and Silvio Micali. "Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing". In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '96. Springer-Verlag, 1996/08/18, pp. 201–215. ISBN: 978-3-540-61512-5.

[Hol97] G.J. Holzmann. "The Model Checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997/05), pp. 279–295. ISSN: 1939-3520. DOI: 10.1109/32.588521. URL: https://ieeexplore.ieee.org/abstract/document/588521.

[Hon+22] Zicong Hong, Song Guo, Rui Zhang, Peng Li, Yufen Zhan, and Wuhui Chen. "Cycle: Sustainable Off-Chain Payment Channel Network with Asynchronous Rebalancing". In: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2022/06, pp. 41–53. DOI: 10.1109/DSN53405.2022.00017.

[How+25] Heidi Howard, Markus A. Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. "Smart Casual Verification of the Confidential Consortium Framework". In: *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*. 22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025. Philadelphia, PA, USA: USENIX Association, 2025, pp. 259–276. ISBN: 978-1-939133-46-5. URL: https://www.usenix.org/conference/nsdi25/presentation/howard.

[HS20] Hans Hüttel and Vilim Staroveški. "Secrecy and Authenticity Properties of the Lightning Network Protocol". In: 6th International Conference on Information Systems Security and Privacy. 2020/02/27, pp. 119–130. ISBN: 978-989-758-399-5. DOI: 10.5220/0008974801190130. URL: https://www.scitepress.org/Link.aspx?doi=10.5220/0008974801190130.

[HS22] Hans Hüttel and Vilim Staroveški. "Key Agreement in the Lightning Network Protocol". In: *Information Systems Security and Privacy*. Ed. by Steven Furnell, Paolo Mori, Edgar Weippl, and Olivier Camp. Communications in Computer and Information Science. Cham: Springer International Publishing, 2022, pp. 139–155. ISBN: 978-3-030-94900-6. DOI: 10.1007/978-3-030-94900-6_7.

[Hua+10] Rong Huang, Yukun Jiang, Zhili Liu, Yuanzhou Yang, and Wen Jiang. "Algorithm and Implementation of Urban Rail Transit Network Based on Joint Operation". In: *Journal of Transportation Systems Engineering and Information Technology* 10.2 (2010/04/01), pp. 130–135. ISSN: 1570-6672. DOI: 10.1016/S1570-6672(09)60038-6. URL: https://www.sciencedirect.com/science/article/pii/S1570667209600386.

[HZ20] Jona Harris and Aviv Zohar. "Flood & Loot: A Systemic Attack on The Lightning Network". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT '20. New York, NY, USA: Association for Computing Machinery, 2020/10/21, pp. 202–213. ISBN: 978-1-4503-8139-0. DOI: 10.1145/3419614.3423248. URL: https://doi.org/10.1145/3419614.3423248.

[JF56] L. R. Ford Jr and D. R. Fulkerson. "Maximal Flow Through a Network". In: *Canadian Journal of Mathematics* 8 (1956/01), pp. 399–404. ISSN: 0008-414X, 1496-4279. DOI: 10.4153/CJM-1956-045-5. URL: https://www.cambridge.org/core/journals/canadian-journal-of-mathematics/article/maximal-flow-through-a-network/5D6E55D3B06C4F7B1043BC1D82D40764.

[JLT20] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. "Lightweight Virtual Payment Channels". In: *Cryptology and Network Security*. Ed. by Stephan Krenn, Haya Shulman, and Serge Vaudenay. Cham: Springer International Publishing, 2020, pp. 365–384. ISBN: 978-3-030-65411-5. DOI: 10.1007/978-3-030-65411-5_18.

[JLT21] Maxim Jourenko, Mario Larangeira, and Keisuke Tanaka. "Payment Trees: Low Collateral Payments for Payment Channel Networks". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2021, pp. 189–208. ISBN: 978-3-662-64331-0. DOI: 10.1007/978-3-662-64331-0_10.

[Jou+19] Maxim Jourenko, Mario Larangeira, Kanta Kurazumi, and Keisuke Tanaka. *SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies*. 354. 2019/04/02, p. 19. URL: https://eprint.iacr.org/2019/352.

[Kap+21] George Kappos, Haaroon Yousaf, Ania Piotrowska, Sanket Kanjalkar, Sergi Delgado-Segura, Andrew Miller, and Sarah Meiklejohn. "An Empirical Analysis of Privacy in the Lightning Network". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer, 2021, pp. 167–186. ISBN: 978-3-662-64322-8. DOI: 10.1007/978-3-662-64322-8_8.

[Kar+09] Dean Karlan, Markus Mobius, Tanya Rosenblat, and Adam Szeidl. "Trust and Social Collateral". In: *The Quarterly Journal of Economics* 124.3 (2009/08/01), pp. 1307–1361. ISSN: 0033-5533. DOI: 10.1162/qjec.2009.124.3.1307. URL: https://doi.org/10.1162/qjec.2009.124.3.1307.

[KAV24] Kartick Kolachala, Mohammed Ababneh, and Roopa Vishwanathan. "SoK: Payment Channel Networks". In: *2024 Conference on Building a Secure & Empowered Cyberspace (BuildSEC)*. 2024 Conference on Building a Secure & Empowered Cyberspace (BuildSEC). 2024/12, pp. 28–35. DOI: 10.1109/BuildSEC64048.2024.00013. URL: https://ieeexplore.ieee.org/abstract/document/10874356.

[KER21] Satwik Prabhu Kumble, Dick Epema, and Stefanie Roos. "How Lightning's Routing Diminishes Its Anonymity". In: *Proceedings of the 16th International Conference on Availability, Reliability and Security*. ARES '21. New York, NY, USA: Association for Computing Machinery, 2021/08/17, pp. 1–10. ISBN: 978-1-4503-9051-4. DOI: 10.1145/3465481.3465761. URL: https://dl.acm.org/doi/10.1145/3465481.3465761.

[KER23] Satwik Prabhu Kumble, Dick Epema, and Stefanie Roos. "Game-Theoretic Analysis of (Non-)Refundable Fees in the Lightning Network". In: *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS). 2023/12, pp. 645–652. DOI: 10.1109/ICPADS60453.2023.00100. URL: https://ieeexplore.ieee.org/document/10476198.

[KG17] Rami Khalil and Arthur Gervais. "Revive: Rebalancing Off-Blockchain Payment Networks". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: ACM, 2017, pp. 439–453. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134033. URL: http://doi.acm.org/10.1145/3133956.3134033.

[KG21] Heba Kadry and Yasser Gadallah. "A Machine Learning-Based Routing Technique for Off-chain Transactions in Payment Channel Networks". In: *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*. 2021 IEEE International Conference on Smart Internet of Things (SmartIoT). 2021/08, pp. 66–73. DOI: 10.1109/SmartIoT52359.2021.00020.

[Kha+18] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. 2018/642. 2018. URL: https://eprint.iacr.org/2018/642.

[KKR24] Julia Khamis, Arad Kotzer, and Ori Rottenstreich. "Topologies for Blockchain Payment Channel Networks: Models and Constructions". In: *IEEE/ACM Transactions on Networking* (2024), pp. 1–17. ISSN: 1558-2566. DOI: 10.1109/TNET.2024.3445274. URL: https://ieeexplore.ieee.org/abstract/document/10679611.

[KNW19] Majid Khabbazian, Tejaswi Nadahalli, and Roger Wattenhofer. "Outpost: A Responsive Lightweight Watchtower". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. AFT '19. New York, NY, USA: ACM, 2019/10/21, pp. 31–40. ISBN: 978-1-4503-6732-5. DOI: 10.1145/3318041.3355464. URL: https://doi.org/10.1145/3318041.3355464.

[Kot+25] Arad Kotzer, Bence Ladóczki, János Tapolcai, and Ori Rottenstreich. "Addressing Scalability Issues of Blockchains With Hypergraph Payment Networks". In: *IEEE Transactions on Network and Service Management* 22.3 (2025/06), pp. 2427–2440. ISSN: 1932-4537. DOI: 10.1109/TNSM.2025.3542960. URL: https://ieeexplore.ieee.org/document/10896831.

[KR21] Julia Khamis and Ori Rottenstreich. "Demand-Aware Channel Topologies for Off-chain Payments". In: *2021 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. 2021 International Conference on COMmunication Systems & NETworkS (COMSNETS). 2021/01, pp. 272–280. DOI: 10.1109/COMSNETS51098.2021.9352899. URL: https://ieeexplore.ieee.org/document/9352899.

[KRK23] Alvi Ataur Khalil, Mohammad Ashiqur Rahman, and Hisham A. Kholidy. "FAKEY: Fake Hashed Key Attack on Payment Channel Networks". In: *2023 IEEE Conference on Communications and Network Security (CNS)*. 2023 IEEE Conference on Communications and Network Security (CNS). 2023/10, pp. 1–9. DOI: 10.1109/CNS59707.2023.10288911. URL: https://ieeexplore.ieee.org/abstract/document/10288911.

[KS20] Nida Khan and Radu State. "Lightning Network: A Comparative Review of Transaction Fees and Data Analysis". In: *Blockchain and Applications*. Ed. by Javier Prieto, Ashok Kumar Das, Stefano Ferretti, António Pinto, and Juan Manuel Corchado. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 11–18. ISBN: 978-3-030-23813-1. DOI: 10.1007/978-3-030-23813-1_2.

[KSE21] Kemal Akkaya, Suat Mercan, and Enes Erdin. "An Evaluation of Cryptocurrency Payment Channel Networks and Their Privacy Implications". In: *ITU Journal on Future and Evolving Technologies* 2.1 (2021/03/15), pp. 35–44. ISSN: 2616-8375. DOI: 10.52953/clbl7402. URL: https://www.itu.int/pub/S-JNL-VOL2.ISSUE1-2021-A03.

[KT20] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. "A Composable Security Treatment of the Lightning Network". In: *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). 2020/06, pp. 334–349. DOI: 10.1109/CSF49147.2020.00031.

[KT21] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. *Elmo: Recursive Virtual Payment Channels for Bitcoin*. 747. 2021. URL: https://eprint.iacr.org/2021/747.

[KZZ16] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. "Fair and Robust Multi-party Computation Using a Global Transaction Ledger". In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer, 2016, pp. 705–734. ISBN: 978-3-662-49896-5. DOI: 10.1007/978-3-662-49896-5_25.

[LA22] Mengling Liu and Man Ho Au. "Practical Anonymous Multi-hop Locks for Lightning Network Compatible Payment Channel Networks". In: *Network and System Security: 16th International Conference, NSS 2022, Denarau Island, Fiji, December 9–12, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022/12/09, pp. 547–560. ISBN: 978-3-031-23019-6. DOI: 10.1007/978-3-031-23020-2_31. URL: https://doi.org/10.1007/978-3-031-23020-2_31.

[Lab26] Talaia Labs. *The Eye of Satoshi (Rust-Teos)*. Talaia Labs, 2026/02/19. URL: https://github.com/talaia-labs/rust-teos.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. 364 pp. ISBN: 978-0-321-14306-8. URL: https://lamport.azurewebsites.net/tla/book.html.

[Lam05] Leslie Lamport. "Real-Time Model Checking Is Really Simple". In: *Correct Hardware Design and Verification Methods*. Ed. by Dominique Borrione and Wolfgang Paul. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 162–175. ISBN: 978-3-540-32030-2. DOI: 10.1007/11560548_14.

[Lam12] Leslie Lamport. "How to Write a 21st Century Proof". In: *Journal of Fixed Point Theory and Applications* 11.1 (2012/03), pp. 43–63. ISSN: 1661-7738, 1661-7746. DOI: 10.1007/s11784-012-0071-6. URL: http://link.springer.com/10.1007/s11784-012-0071-6.

[Lam94] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Transactions on Programming Languages and Systems* 16.3 (1994/05/01), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: https://doi.org/10.1145/177492.177726.

[Law07] Averill M. Law. *Simulation Modeling And Analysis*. Boston: McGraw-Hill, 2007. 800 pp. ISBN: 978-0-07-066733-4. URL: https://www.averill-law.com/simulation-book/.

[Lee+24] Junmo Lee, Seongjun Kim, Sanghyeon Park, and Soo-Mook Moon. "RouTEE: Secure, Scalable, and Efficient Off-Chain Payments Using Trusted Execution Environments". In: *2024 Annual Computer Security Applications Conference (ACSAC)*. 2024 Annual Computer Security Applications Conference (ACSAC). 2024/12, pp. 456–472. DOI: 10.1109/ACSAC63791.2024.00048. URL: https://ieeexplore.ieee.org/document/10917388.

[Lei+19] Marc Leinweber, Matthias Grundmann, Leonard Schönborn, and Hannes Hartenstein. "TEE-Based Distributed Watchtowers for Fraud Protection in the Lightning Network". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro. Vol. 11737. Lecture Notes in Computer Science. Springer International Publishing, 2019/09, pp. 177–194. ISBN: 978-3-030-31500-9.

[LHY22] Zhichun Lu, Runchao Han, and Jiangshan Yu. "General Congestion Attack on HTLC-Based Payment Channel Networks". In: *DROPS-IDN/v2/Document/10.4230/OASIcs.Tokenomics.2021.2*. 3rd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2021). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/OASIcs.Tokenomics.2021.2. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Tokenomics.2021.2.

[Li+21] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang, and Xianghong Liu. "TopoShot: Uncovering Ethereum's Network Topology Leveraging Replacement Transactions". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021/11/02, pp. 302–319. ISBN: 978-1-4503-9129-0. DOI: 10.1145/3487552.3487814. URL: https://doi.org/10.1145/3487552.3487814.

[Li+23] Yuxian Li, Jian Weng, Wei Wu, Ming Li, Yingjiu Li, Haoxin Tu, Yongdong Wu, and Robert H. Deng. "PRI: PCH-based Privacy-Preserving with Reusability and Interoperability for Enhancing Blockchain Scalability". In: *Journal of Parallel and Distributed Computing* 180 (2023/10/01), p. 104721. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2023.104721. URL: https://www.sciencedirect.com/science/article/pii/S0743731523000916.

[Lin+19] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. "Teechain: A Secure Payment Network with Asynchronous Blockchain Access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles - SOSP '19*. The 27th ACM Symposium. Huntsville, Ontario, Canada: ACM Press, 2019, pp. 63–79. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359627. URL: http://dl.acm.org/citation.cfm?doid=3341301.3359627.

[Liu+23] Ya Liu, Yuanhang Wu, Fengyv Zhao, and Yanli Ren. "Balanced Off-Chain Payment Channel Network Routing Strategy Based On Weight Calculation". In: *The Computer Journal* (2023/04/05), bxad029. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxad029. URL: https://doi.org/10.1093/comjnl/bxad029.

[LM17] Leslie Lamport and Stephan Merz. "Auxiliary Variables in TLA+". In: (2017/05/27), p. 71.

[Lot+22] Ayelet Lotem, Sarah Azouvi, Patrick McCorry, and Aviv Zohar. "Sliding Window Challenge Process for Congestion Detection". In: *Financial Cryptography and Data Security*. Ed. by Ittay Eyal and Juan Garay. Cham: Springer International Publishing, 2022, pp. 512–530. ISBN: 978-3-031-18283-9. DOI: 10.1007/978-3-031-18283-9_25.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. "Uppaal in a Nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1–2 (1997/12), pp. 134–152. ISSN: 1433-2779. DOI: 10.1007/s100090050010. URL: http://link.springer.com/10.1007/s100090050010.

[LRT21] Kimberly Lange, Elias Rohrer, and Florian Tschorsch. "On the Impact of Attachment Strategies for Payment Channel Networks". In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2021/05, pp. 1–9. DOI: 10.1109/ICBC51069.2021.9461104. URL: https://ieeexplore.ieee.org/abstract/document/9461104.

[LSS20] Bowen Liu, Pawel Szalachowski, and Siwei Sun. "Fail-Safe Watchtowers and Short-lived Assertions for Payment Channels". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020/10/05, pp. 506–518. ISBN: 978-1-4503-6750-9. DOI: 10.1145/3320269.3384716. URL: https://doi.org/10.1145/3320269.3384716.

[LV96] Nancy Lynch and Frits Vaandrager. "Forward and Backward Simulations: II. Timing-Based Systems". In: *Information and Computation* 128.1 (1996/07/10), pp. 1–25. ISSN: 0890-5401. DOI: 10.1006/inco.1996.0060. URL: https://www.sciencedirect.com/science/article/pii/S0890540196900607.

[Mal+17a] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. "SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks". In: Internet Society, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23448. URL: https://www.ndss-symposium.org/ndss2017/ndss-

2017-programme/silentwhispers-enforcing-security-and-privacy-decentralized-credit-networks/.

[Mal+17b] 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. New York, NY, USA: ACM, 2017, pp. 455–471. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134096. URL: http://doi.acm.org/10.1145/3133956.3134096.

[Mal+19] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23330. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_09-4_Malavolta_paper.pdf.

[Mat+17] Sinisa Matetic, David Sommer, Mansoor Ahmed, Arthur Gervais, Kari Kostiainen, Aritra Dhar, Ari Juels, and Srdjan Capkun. "ROTE: Rollback Protection for Trusted Execution". In: (2017/08), p. 19.

[MBR20] S. Mazumdar, P. Banerjee, and S. Ruj. "Time Is Money: Countering Griefing Attack in Lightning Network". In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). 2020/12, pp. 1036–1043. DOI: 10.1109/TrustCom50675.2020.00138.

[McC+16] Patrick McCorry, Malte Möser, Siamak F. Shahandasti, and Feng Hao. "Towards Bitcoin Payment Networks". In: *Information Security and Privacy*. Ed. by Joseph K. Liu and Ron Steinfeld. Vol. 9722. Cham: Springer International Publishing, 2016, pp. 57–76. ISBN: 978-3-319-40252-9 978-3-319-40253-6. DOI: 10.1007/978-3-319-40253-6_4. URL: http://link.springer.com/10.1007/978-3-319-40253-6_4.

[McC+19] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, and Andrew Miller. "Pisa: Arbitration Outsourcing for State Channels". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. AFT '19. New York, NY, USA: ACM, 2019/10/21, pp. 16–30. ISBN: 978-1-4503-6732-5. DOI: 10.1145/3318041.3355461. URL: https://doi.org/10.1145/3318041.3355461.

[McK+16] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. HASP '16. New York, NY, USA: Association for Computing Machinery, 2016/06/18, pp. 1–9. ISBN: 978-1-4503-4769-3. DOI: 10.1145/

2948618.2954331. URL: https://dl.acm.org/doi/10.1145/2948618.2954331.

[Mei+13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. "The TAMARIN Prover for the Symbolic Analysis of Security Protocols". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer, 2013, pp. 696–701. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_48.

[MF20] Stefano Martinazzi and Andrea Flori. "The Evolving Topology of the Lightning Network: Centralization, Efficiency, Robustness, Synchronization, and Anonymity". In: *PLOS ONE* 15.1 (2020/01/15), e0225966. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0225966. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0225966.

[Mil+15] Andrew Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. *Discovering Bitcoin's Public Topology and Influential Nodes*. 2015.

[Mil+19] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. "Sprites and State Channels: Payment Networks That Go Faster Than Lightning". In: *Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 508–526. ISBN: 978-3-030-32101-7. DOI: 10.1007/978-3-030-32101-7_30.

[Mir+21] Arash Mirzaei, Amin Sakzad, Jiangshan Yu, and Ron Steinfeld. "FPPW: A Fair and Privacy Preserving Watchtower for Bitcoin". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer, 2021, pp. 151–169. ISBN: 978-3-662-64331-0. DOI: 10.1007/978-3-662-64331-0_8.

[Mir+22] Arash Mirzaei, Amin Sakzad, Jiangshan Yu, and Ron Steinfeld. "Garrison: A Novel Watchtower Scheme for Bitcoin". In: *Information Security and Privacy*. Ed. by Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo. Cham: Springer International Publishing, 2022, pp. 489–508. ISBN: 978-3-031-22301-3. DOI: 10.1007/978-3-031-22301-3_24.

[Mis+08] Alan Mislove, Ansley Post, Peter Druschel, and Krishna P. Gummadi. "Ostra: Leveraging Trust to Thwart Unwanted Communication". In: 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08). 2008. URL: https://www.usenix.org/conference/nsdi-08/ostra-leveraging-trust-thwart-unwanted-communication.

[Mor+20] Pedro Moreno-Sanchez, Arthur Blue, Duc V. Le, Sarang Noether, Brandon Goodell, and Aniket Kate. "DLSAG: Non-interactive Refund Transactions for Interoperable Payment Channels in Monero". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Cham: Springer International Publishing, 2020, pp. 325–345. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_18.

[MR23] Subhra Mazumdar and Sushmita Ruj. "CryptoMaze: Privacy-Preserving Splitting of Off-Chain Payments". In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (2023/03), pp. 1060–1073. ISSN: 1941-0018. DOI: 10.1109/TDSC.2022.3148476. URL: https://ieeexplore.ieee.org/document/9705551.

[MZ21a] Ayelet Mizrahi and Aviv Zohar. "Congestion Attacks in Payment Channel Networks". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer, 2021, pp. 170–188. ISBN: 978-3-662-64331-0. DOI: 10.1007/978-3-662-64331-0_9.

[MZ21b] Ayelet Mizrahi and Aviv Zohar. "Congestion Attacks in Payment Channel Networks". In: *ArXiv200206564 Cs* (2021/01/31). arXiv: 2002.06564 [cs]. URL: http://arxiv.org/abs/2002.06564.

[NAH16] T. Neudecker, P. Andelfinger, and H. Hartenstein. "Timing Analysis for Inferring the Topology of the Bitcoin Peer-to-Peer Network". In: *Proceedings of the 13th IEEE International Conference on Advanced and Trusted Computing*. 13th IEEE International Conference on Advanced and Trusted Computing. 2016/07, pp. 358–367. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0070.

[Nak08] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008, p. 9. URL: https://bitcoin.org/bitcoin.pdf.

[Neu19] Till Neudecker. "Security and Anonymity Aspects of the Network Layer of Permissionless Blockchains". 2019. DOI: 10.5445/IR/1000089033. URL: https://publikationen.bibliothek.kit.edu/1000089033.

[NKW21] Tejaswi Nadahalli, Majid Khabbazian, and Roger Wattenhofer. "Timelocked Bribing". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Berlin, Heidelberg: Springer, 2021, pp. 53–72. ISBN: 978-3-662-64322-8. DOI: 10.1007/978-3-662-64322-8_3.

[NT24] Charmaine Ndolo and Florian Tschorsch. "On the (Not So) Surprising Impact of Multi-Path Payments on Performance And Privacy in the Lightning Network". In: *Computer Security. ESORICS 2023 International Workshops*. Ed. by Sokratis Katsikas, Frédéric Cuppens, Nora Cuppens-Boulahia, Costas Lambrinoudakis, Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Pantaleone Nespoli, Christos Kalloniatis, John Mylopoulos, Annie Antón, and Stefanos Gritzalis. Cham: Springer Nature Switzerland, 2024, pp. 411–427. ISBN: 978-3-031-54204-6. DOI: 10.1007/978-3-031-54204-6_25.

[OCZ15] Adegboyega Ojo, Edward Curry, and Fatemeh Ahmadi Zeleti. "A Tale of Open Data Innovations in Five Smart Cities". In: *2015 48th Hawaii International Conference on System Sciences*. 2015 48th Hawaii International Conference on System Sciences. 2015/01, pp. 2326–2335. DOI: 10.1109/HICSS.2015.280.

[OPH22] Luis E. Oleas-Chávez, Cristina Pérez-Solà, and Jordi Herrera-Joancomartí. "Apples and Oranges: On How to Measure Node Centrality in Payment Channel Networks". In: *IEEE Access* 10 (2022), pp. 55469–55487. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3174549.

[Pan+19] Chen Pan, Shuyang Tang, Zhonghui Ge, Zhiqiang Liu, Yu Long, Zhen Liu, and Dawu Gu. "Gnocchi: Multiplexed Payment Channels for Cryptocurrencies". In: *Network and System Security*. International Conference on Network and System Security. Springer, Cham, 2019/12/15, pp. 488–503. DOI: 10.1007/978-3-030-36938-5_30. URL: https://link.springer.com/chapter/10.1007/978-3-030-36938-5_30.

[Pap+18] Giuseppe Pappalardo, Tiziana Di Matteo, Guido Caldarelli, and Tomaso Aste. "Blockchain Inefficiency in the Bitcoin Peers Network". In: *EPJ Data Science* 7.1 (2018/09/05), p. 30. ISSN: 2193-1127. DOI: 10.1140/epjds/s13688-018-0159-3. URL: https://doi.org/10.1140/epjds/s13688-018-0159-3.

[Par+18] Yussef Parcianello, Nádia P. Kozievitch, Keiko V. O. Fonseca, Marcelo de O. Rosa, Tatiana M. C. Gadda, and Francisco C. Malucelli. "Transportation: An Overview from Open Data Approach". In: *2018 IEEE International Smart Cities Conference (ISC2)*. 2018 IEEE International Smart Cities Conference (ISC2). 2018/09, pp. 1–8. DOI: 10.1109/ISC2.2018.8656937.

[Par+19] Sehyun Park, Seongwon Im, Youhwan Seol, and Jeongyeup Paek. "Nodes in the Bitcoin Network: Comparative Measurement Study and Survey". In: *IEEE Access* 7 (2019), pp. 57009–57022. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2914098.

[PD16] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. 2016, p. 59. URL: https://lightning.network/lightning-network-paper.pdf.

[Pér+20] Cristina Pérez-Solà, Alejandro Ranchal-Pedrosa, Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, and Joaquin Garcia-Alfaro. "LockDown: Balance Availability Attack Against Lightning Network Channels". In: *Financial Cryptography and Data Security*. Ed. by Joseph Bonneau and Nadia Heninger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 245–263. ISBN: 978-3-030-51280-4. DOI: 10.1007/978-3-030-51280-4_14.

[Pic+21] Rene Pickhardt, Sergei Tikhomirov, Alex Biryukov, and Mariusz Nowostawski. "Security and Privacy of Lightning Network Payments with Uncertain Channel Balances". In: *ArXiv210308576 Cs* (2021/03/15). arXiv: 2103.08576 [cs]. URL: http://arxiv.org/abs/2103.08576.

[PMZ21] Bernd Prünster, Alexander Marsalek, and Thomas Zefferer. "Total Eclipse of the Heart – Disrupting the InterPlanetary File System". In: *31st USENIX Security Symposium* (2021/09/03). URL: https://blog.ipfs.io/2020-10-30-dht-hardening/.

[PN18] Dmytro Piatkivskyi and Mariusz Nowostawski. "Split Payments in Payment Networks". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 67–75. ISBN: 978-3-030-00305-0.

[PN20] Rene Pickhardt and Mariusz Nowostawski. "Imbalance Measure and Proactive Channel Rebalancing Algorithm for the Lightning Network". In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2020/05, pp. 1–5. DOI: 10.1109/ICBC48266.2020.9169456. URL: https://ieeexplore.ieee.org/document/9169456.

[POR24] Yekaterina Podiatchev, Ariel Orda, and Ori Rottenstreich. "Survivable Payment Channel Networks". In: *IEEE Transactions on Network and Service Management* 21.6 (2024/12), pp. 6218–6232. ISSN: 1932-4537. DOI: 10.1109/TNSM.2024.3456229. URL: https://ieeexplore.ieee.org/abstract/document/10673999.

[PPT19] Alejandro Ranchal Pedrosa, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. "Scalable Lightning Factories for Bitcoin". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC '19. Limassol, Cyprus: Association for Computing Machinery, 2019/04/08, pp. 302–309. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297312. URL: https://doi.org/10.1145/3297280.3297312.

[PR05] Christos H. Papadimitriou and David Ratajczak. "On a Conjecture Related to Geometric Routing". In: *Theoretical Computer Science*. Algorithmic Aspects of Wireless Sensor Networks 344.1 (2005/11/11), pp. 3–14. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2005.06.022. URL: https://www.sciencedirect.com/science/article/pii/S0304397505003725.

[PT20] N. Papadis and L. Tassiulas. "Blockchain-Based Payment Channel Networks: Challenges and Recent Advances". In: *IEEE Access* 8 (2020), pp. 227596–227609. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3046020.

[PT22] Nikolaos Papadis and Leandros Tassiulas. "Payment Channel Networks: Single-Hop Scheduling for Throughput Maximization". In: *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. IEEE INFOCOM 2022 - IEEE Conference on Computer Communications. 2022/05, pp. 900–909. DOI: 10.1109/INFOCOM48880.2022.9796862.

[Qi+25] Minfeng Qi, Qin Wang, Zhipeng Wang, Manvir Schneider, Tianqing Zhu, Shiping Chen, William Knottenbelt, and Thomas Hardjono. "SoK: Bitcoin Layer Two (L2)". In: *ACM Comput. Surv.* 58.3 (2025/09/29), 76:1–76:37. ISSN: 0360-0300. DOI: 10.1145/3763232. URL: https://dl.acm.org/doi/10.1145/3763232.

[Qin+23] Xianrui Qin, Shimin Pan, Arash Mirzaei, Zhimei Sui, Oğuzhan Ersoy, Amin Sakzad, Muhammed F. Esgin, Joseph K. Liu, Jiangshan Yu, and Tsz Hon Yuen. "BlindHub: Bitcoin-Compatible Privacy-Preserving Payment Channel Hubs Supporting Variable Amounts". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023 IEEE Symposium on Security and Privacy (SP). 2023/05, pp. 2462–2480. DOI: 10.1109/SP46215.2023.10179427. URL: https://ieeexplore.ieee.org/abstract/document/10179427.

[Rai+23] Sophie Rain, Georgia Avarikioti, Laura Kovács, and Matteo Maffei. "Towards a Game-Theoretic Security Analysis of Off-Chain Protocols". In: *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. 2023 IEEE 36th Computer Security Foundations Symposium (CSF). 2023/07, pp. 107–122. DOI: 10.1109/CSF57540.2023.00003. URL: https://ieeexplore.ieee.org/abstract/document/10221879.

[Rin86] Dan B. Rinks. "Revenue Allocation Methods for Integrated Transit Systems". In: *Transportation Research Part A: General* 20.1 (1986/01/01), pp. 39–50. ISSN: 0191-2607. DOI: 10.1016/0191-2607(86)90014-2. URL: https://www.sciencedirect.com/science/article/pii/0191260786900142.

[RK22] Sonbol Rahimpour and Majid Khabbazian. "Torrent: Strong, Fast Balance Discovery in the Lightning Network". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2022/05, pp. 1–7. DOI: 10.1109/ICBC54727.2022.9805520.

[RLT17] Elias Rohrer, Jann-Frederik Laß, and Florian Tschorsch. "Towards a Concurrent and Distributed Route Selection for Payment Channel Networks". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Ed. by Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 411–419. ISBN: 978-3-319-67816-0.

[RMT19] Elias Rohrer, Julian Malliaris, and Florian Tschorsch. "Discharged Payment Channels: Quantifying the Lightning Network's Resilience to Topology-Based Attacks". In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*. 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW). 2019/06, pp. 347–356. DOI: 10.1109/EuroSPW.2019.00045.

[RN21] Antoine Riard and Gleb Naumenko. "Time-Dilation Attacks on the Lightning Network". In: *Cryptoeconomic Systems* 1.2 (2021/10/22). ISSN: 2767-4207, DOI: 10.21428/58320208.6ac6960a. URL: https://cryptoeconomicsystems.pubpub.org/pub/riard-lightning-dilation/release/1.

[Rom+21] Matteo Romiti, Friedhelm Victor, Pedro Moreno-Sanchez, Peter Sebastian Nordholt, Bernhard Haslhofer, and Matteo Maffei. "Cross-Layer Deanonymization Methods in the Lightning Protocol". In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2021, pp. 187–204. ISBN: 978-3-662-64322-8. DOI: 10.1007/978-3-662-64322-8_9.

[Roo+17] Stefanie Roos, Pedro Moreno-Sanchez, Aniket Kate, and Ian Goldberg. "Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions". In: *ArXiv170905748 Cs* (2017/09/17). arXiv: 1709.05748 [cs]. URL: http://arxiv.org/abs/1709.05748.

[RT20] Elias Rohrer and Florian Tschorsch. "Counting Down Thunder: Timing Attacks on Privacy in Payment Channel Networks". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT '20. New York, NY, USA: Association for Computing Machinery, 2020/10/21, pp. 214–227. ISBN: 978-1-4503-8139-0. DOI: 10.1145/3419614.3423262. URL: https://doi.org/10.1145/3419614.3423262.

[Rus19] Rusty Russell. *[Lightning-dev] Full Disclosure: CVE-2019-12998 / CVE-2019-12999 / CVE-2019-13000*. E-mail. 2019/09/27. URL: https://diyhpl.us/~bryan/irc/bitcoin/bitcoin-dev/linuxfoundation-pipermail/lightning-dev/2019-September/002174.txt.

[Sch19] Leonard Schönborn. "Betrugsschutz Für Das Lightning-Netzwerk Mit Intel SGX". Karlsruhe Institute of Technology, 2019/05/31.

[Ser+20] István András Seres, László Gulyás, Dániel A. Nagy, and Péter Burcsi. "Topological Analysis of Bitcoin's Lightning Network". In: *Mathematical Research for Blockchain Economy*. Ed. by Panos Pardalos, Ilias Kotsireas, Yike Guo, and William Knottenbelt. Springer Proceedings in Business and Economics. Cham: Springer International Publishing, 2020, pp. 1–12. ISBN: 978-3-030-37110-4. DOI: 10.1007/978-3-030-37110-4_1.

[SK24] Neeraj Sharma and Kalpesh Kapoor. "maxREE: Maximizing Flow by Replacing Exhausted Edges in Lightning Network". In: *IEEE Transactions on Network Science and Engineering* (2024), pp. 1–15. ISSN: 2327-4697. DOI: 10.1109/TNSE.2024.3522198. URL: https://ieeexplore.ieee.org/abstract/document/10816058.

[SS23] Cosimo Sguanci and Anastasios Sidiropoulos. "Mass Exit Attacks on the Lightning Network". In: *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2023/05, pp. 1–3. DOI: 10.1109/ICBC56567.2023.10174926. URL: https://ieeexplore.ieee.org/document/10174926.

[Sui+22] Zhimei Sui, Joseph K. Liu, Jiangshan Yu, and Xianrui Qin. "MoNet: A Fast Payment Channel Network for Scriptless Cryptocurrency Monero". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS). 2022/07, pp. 280–290. DOI: 10.1109/ICDCS54860.2022.00035. URL: https://ieeexplore.ieee.org/abstract/document/9912301.

[SZT22a] Yahya Shahsavari, Kaiwen Zhang, and Chamseddine Talhi. "A Theoretical Model for Block Propagation Analysis in Bitcoin Network". In: *IEEE Transactions on Engineering Management* 69.4 (2022/08), pp. 1459–1476. ISSN: 1558-0040. DOI: 10.1109/TEM.2020.2989170. URL: https://ieeexplore.ieee.org/document/9099002/?arnumber=9099002.

[SZT22b] Yahya Shahsavari, Kaiwen Zhang, and Chamseddine Talhi. "Toward Quantifying Decentralization of Blockchain Networks With Relay Nodes". In: *Frontiers in Blockchain* 5 (2022). ISSN: 2624-7852. URL: https://www.frontiersin.org/article/10.3389/fbloc.2022.812957.

[TA06] Dimitrios A. Tsamboulas and Constantinos Antoniou. "Allocating Revenues to Public Transit Operators under an Integrated Fare System". In: *Transportation Research Record* 1986.1 (2006/01/01), pp. 29–37. ISSN: 0361-1981. DOI: 10.1177/0361198106198600104. URL: https://doi.org/10.1177/0361198106198600104.

[Tan+20] Weizhao Tang, Weina Wang, Giulia Fanti, and Sewoong Oh. "Privacy-Utility Tradeoffs in Routing Cryptocurrency over Payment Channel Networks". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.2 (2020/06/12), 29:1–29:39. DOI: 10.1145/3392147. URL: https://dl.acm.org/doi/10.1145/3392147.

[Tao+23] Yuechen Tao, Bo Li, Baochun Li, and Lei Chen. "Characterizing Performance Limits in Payment Channel Networks". In: *IEEE Transactions on Knowledge and Data Engineering* 35.3 (2023/03), pp. 2353–2365. ISSN: 1558-2191. DOI: 10.1109/TKDE.2021.3120842. URL: https://ieeexplore.ieee.org/abstract/document/9580733.

[TB20] S. Thakur and J. G. Breslin. "Coordinated Landmark-based Routing for Blockchain Offline Channels". In: *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*. 2020 Second International Conference on Blockchain Computing and Applications (BCCA). 2020/11, pp. 108–114. DOI: 10.1109/BCCA50787.2020.9274462.

[Tea24b] The Tamarin Team. *Tamarin-Prover Manual*. 2024. URL: https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf.

[Thy+22] Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Fritz Schmid, and Dominique Schröder. "Verifiable Timed Linkable Ring Signatures for Scalable Payments for Monero". In: *Computer Security – ESORICS 2022*. Ed. by Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng. Cham: Springer Nature Switzerland, 2022, pp. 467–486. ISBN: 978-3-031-17146-8. DOI: 10.1007/978-3-031-17146-8_23.

[Tik+20] Sergei Tikhomirov, Rene Pickhardt, Alex Biryukov, and Mariusz Nowostawski. "Probing Channel Balances in the Lightning Network". In: *ArXiv200400333 Cs* (2020/04/01). arXiv: 2004.00333 [cs]. URL: http://arxiv.org/abs/2004.00333.

[Tiw+23] Samarth Tiwari, Michelle Yeo, Zeta Avarikioti, Iosif Salem, Krzysztof Pietrzak, and Stefan Schmid. "Wiser: Increasing Throughput in Payment Channel Networks with Transaction Aggregation". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. AFT '22. New York, NY, USA: Association for Computing Machinery, 2023/07/05, pp. 217–231. ISBN:

978-1-4503-9861-9. DOI: 10.1145/3558535.3559775. URL: https://dl.acm.org/doi/10.1145/3558535.3559775.

[TM20] Somanath Tripathy and Susil Kumar Mohanty. "MAPPCN: Multi-hop Anonymous and Privacy-Preserving Payment Channel Network". In: *Financial Cryptography and Data Security*. Ed. by Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 481–495. ISBN: 978-3-030-54455-3. DOI: 10.1007/978-3-030-54455-3_34.

[TMM20] Sergei Tikhomirov, Pedro Moreno-Sanchez, and Matteo Maffei. "A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network". In: *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). 2020/09, pp. 387–396. DOI: 10.1109/EuroSPW51379.2020.00059. URL: https://ieeexplore.ieee.org/abstract/document/9229723.

[TMM21] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. "A2L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). 2021/05, pp. 1834–1851. DOI: 10.1109/SP40001.2021.00111. URL: https://ieeexplore.ieee.org/abstract/document/9519431.

[Tra+22] Tuan Tran, Haofan Zheng, Peter Alvaro, and Owen Arden. "Payment Channels Under Network Congestion". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). 2022/05, pp. 1–5. DOI: 10.1109/ICBC54727.2022.9805547.

[Tsa+21] Itay Tsabary, Matan Yechieli, Alex Manuskin, and Ittay Eyal. "MAD-HTLC: Because HTLC Is Crazy-Cheap to Attack". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021 IEEE Symposium on Security and Privacy (SP). 2021/05, pp. 1230–1248. DOI: 10.1109/SP40001.2021.00080. URL: https://ieeexplore.ieee.org/abstract/document/9519476.

[TSH22] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. "Blockchain Scaling Using Rollups: A Comprehensive Survey". In: *IEEE Access* 10 (2022), pp. 93039–93054. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3200051. URL: https://ieeexplore.ieee.org/document/9862815.

[Tsu88] P. F. Tsuchiya. "The Landmark Hierarchy: A New Hierarchy for Routing in Very Large Networks". In: *SIGCOMM Comput. Commun. Rev.* 18.4 (1988/08/01), pp. 35–42. ISSN: 0146-4833. DOI: 10.1145/52325.52329. URL: https://dl.acm.org/doi/10.1145/52325.52329.

[Tuv+22] Giovanni Tuveri, Marco Garau, Eleonora Sottile, Lucia Pintor, Luigi Atzori, and Italo Meloni. "Beep4Me: Automatic Ticket Validation to Support Fare Clearing and Service Planning". In: *Sensors* 22.4 (2022/01), p. 1543. ISSN:

1424-8220. DOI: 10.3390/s22041543. URL: https://www.mdpi.com/1424-8220/22/4/1543.

[TVV23] Mohammad Hossein Tabatabaei, Roman Vitenberg, and Narasimha Raghavan Veeraragavan. "Understanding Blockchain: Definitions, Architecture, Design, and System Comparison". In: *Computer Science Review* 50 (2023/11/01), p. 100575. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2023.100575. URL: https://www.sciencedirect.com/science/article/pii/S1574013723000424.

[TZS20] Saar Tochner, Aviv Zohar, and Stefan Schmid. "Route Hijacking and DoS in Off-Chain Networks". In: *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*. AFT '20. New York, NY, USA: Association for Computing Machinery, 2020/10/26, pp. 228–240. ISBN: 978-1-4503-8139-0. DOI: 10.1145/3419614.3423253. URL: https://dl.acm.org/doi/10.1145/3419614.3423253.

[Var25a] Various. *Apalache*. 2025. URL: https://apalache-mc.org/.

[Var25b] Various. *TLC Model Checker*. TLA+, 2025/09/17. URL: https://github.com/tlaplus/tlaplus.

[vDam+20] Gijs van Dam, Rabiah Abdul Kadir, Puteri N. E. Nohuddin, and Halimah Badioze Zaman. "Improvements of the Balance Discovery Attack on Lightning Network Payment Channels". In: *ICT Systems Security and Privacy Protection*. Ed. by Marko Hölbl, Kai Rannenberg, and Tatjana Welzer. Cham: Springer International Publishing, 2020, pp. 313–323. ISBN: 978-3-030-58201-2. DOI: 10.1007/978-3-030-58201-2_21.

[Vis+12] Bimal Viswanath, Mainack Mondal, Krishna P. Gummadi, Alan Mislove, and Ansley Post. "Canal: Scaling Social Network-Based Sybil Tolerance Schemes". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. New York, NY, USA: Association for Computing Machinery, 2012/04/10, pp. 309–322. ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168867. URL: https://dl.acm.org/doi/10.1145/2168836.2168867.

[Vis25] Visa. *Visa Annual Report 2025*. 2025/12/08. URL: https://s29.q4cdn.com/385744025/files/doc_downloads/2025/Visa-Fiscal-2025-Annual-Report.pdf.

[VK25] Danila Valko and Daniel Kudenko. "Hybrid Pathfinding Optimization for the Lightning Network with Reinforcement Learning". In: *Engineering Applications of Artificial Intelligence* 146 (2025/04/15), p. 110225. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2025.110225. URL: https://www.sciencedirect.com/science/article/pii/S0952197625002258.

[Wan+19a] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. "SoK: Sharding on Blockchain". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (Zurich, Switzerland). AFT '19. New York, NY, USA: ACM, 2019, pp. 41–61. ISBN: 978-1-4503-6732-5. DOI: 10.1145/3318041.3355457. URL: http://doi.acm.org/10.1145/3318041.3355457.

[Wan+19b] Peng Wang, Hong Xu, Xin Jin, and Tao Wang. "Flash: Efficient Dynamic Routing for Offchain Networks". In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies - CoNEXT '19*. The 15th International Conference. Orlando, Florida: ACM Press, 2019, pp. 370–381. ISBN: 978-1-4503-6998-5. DOI: 10.1145/3359989.3365411. URL: http://dl.acm.org/citation.cfm?doid=3359989.3365411.

[Wan+21] Taotao Wang, Chonghe Zhao, Qing Yang, Shengli Zhang, and Soung Chang Liew. "Ethna: Analyzing the Underlying Peer-to-Peer Network of Ethereum Blockchain". In: *IEEE Transactions on Network Science and Engineering* 8.3 (2021/07), pp. 2131–2146. ISSN: 2327-4697. DOI: 10.1109/TNSE.2021.3078181.

[Wan+24] Jianhuan Wang, Shang Gao, Guyue Li, Keke Gai, and Bin Xiao. "SAMCU: Secure and Anonymous Multi-Channel Updates in Payment Channel Networks". In: *IEEE Transactions on Information Forensics and Security* 19 (2024), pp. 9115–9128. ISSN: 1556-6021. DOI: 10.1109/TIFS.2024.3451366. URL: https://ieeexplore.ieee.org/abstract/document/10654379.

[Wan+25] Xiaojie Wang, Hanxue Li, Ling Yi, Zhaolong Ning, Xiaoming Tao, Song Guo, and Yan Zhang. "A Survey on Off-chain Networks: Frameworks, Technologies, Solutions and Challenges". In: *ACM Comput. Surv.* (2025/05/08). ISSN: 0360-0300. DOI: 10.1145/3735124. URL: https://dl.acm.org/doi/10.1145/3735124.

[Wei+24] Ben Weintraub, Satwik Prabhu Kumble, Cristina Nita-Rotaru, and Stefanie Roos. "Payout Races and Congested Channels: A Formal Analysis of Security in the Lightning Network". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024/12/09, pp. 2562–2576. ISBN: 979-8-4007-0636-3. DOI: 10.1145/3658644.3670315. URL: https://dl.acm.org/doi/10.1145/3658644.3670315.

[Wen+25] Hongbo Wen, Hanzhi Liu, Jingyu Ke, Yanju Chen, Dahlia Malkhi, and Yu Feng. *Thunderbolt: A Formally Verified Protocol for Off-Chain Bitcoin Transfers*. 2025/709. 2025. URL: https://eprint.iacr.org/2025/709.

[WH20] Finnegan Waugh and Ralph Holz. "An Empirical Study of Availability and Reliability Properties of the Bitcoin Lightning Network". In: *ArXiv200614358 Cs* (2020/06/25). arXiv: 2006.14358 [cs]. URL: http://arxiv.org/abs/2006.14358.

[WJ22] Jie Wu and Suhan Jiang. "On Increasing Scalability and Liquidation of Lightning Networks for Blockchains". In: *IEEE Transactions on Network Science and Engineering* 9.4 (2022/07), pp. 2589–2600. ISSN: 2327-4697. DOI: 10.1109/TNSE.2022.3166707. URL: https://ieeexplore.ieee.org/abstract/document/9763398.

[WP17] Liang Wang and Ivan Pustogarov. "Towards Better Understanding of Bitcoin Unreachable Peers". In: *ArXiv170906837 Cs* (2017/09/20). arXiv: 1709.06837 [cs]. URL: http://arxiv.org/abs/1709.06837.

[WRW24] Caimei Wang, Yudong Ren, and Zhize Wu. "Multi-Hop Anonymous Payment Channel Network Based on Onion Routing". In: *IET Blockchain* 4.2 (2024), pp. 197–208. ISSN: 2634-1573. DOI: 10.1049/blc2.12065. URL: https://onlinelibrary.wiley.com/doi/abs/10.1049/blc2.12065.

[Xia+17] Huali Xiao, Li Sun, Shaohong Kong, Guiwei Gong, and Fan Zhang. "Passenger Travel Path Estimation Algorithm Based on High Accuracy Location Data". In: *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*. 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD). 2017/08, pp. 256–260. DOI: 10.1109/CBD.2017.51.

[Xie+23] Songyou Xie, Lijun Xiao, Dezhi Han, Kun Xie, Xiong Li, and Wei Liang. "HCVC: A High-Capacity Off-Chain Virtual Channel Scheme Based on Bidirectional Locking Mechanism". In: *IEEE Transactions on Network Science and Engineering* (2023), pp. 1–12. ISSN: 2327-4697. DOI: 10.1109/TNSE.2023.3332130. URL: https://ieeexplore.ieee.org/document/10321707?dld=Z2xhc21haWwuZGU%3D&source=SEARCHALERT.

[XZZ24] Minze Xu, Yuan Zhang, and Sheng Zhong. "Towards Payment Channel Watchtowers with Collateral-free Security and Robustness". In: *IEEE Transactions on Dependable and Secure Computing* (2024), pp. 1–18. ISSN: 1941-0018. DOI: 10.1109/TDSC.2024.3445429. URL: https://ieeexplore.ieee.org/abstract/document/10638799.

[Yan+23] Lingxiao Yang, Xuewen Dong, Sheng Gao, Qiang Qu, Xiaodong Zhang, Wensheng Tian, and Yulong Shen. "Optimal Hub Placement and Deadlock-Free Routing for Payment Channel Network Scalability". In: *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. 2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS). 2023/07, pp. 692–702. DOI: 10.1109/ICDCS57875.2023.00087. URL: https://ieeexplore.ieee.org/document/10272518.

[Ye+19] YongJie Ye, Jingjing Zhang, Weigang Wu, Xiapu Luo, and Jiannong Cao. "Boros: Secure Cross-Channel Transfers via Channel Hub". In: *ArXiv191112929 Cs* (2019/11/28). arXiv: 1911.12929 [cs]. URL: http://arxiv.org/abs/1911.12929.

[Ye+21] Yongjie Ye, Zhifeng Ren, Xiapu Luo, Jingjing Zhang, and Weigang Wu. "Garou: An Efficient and Secure Off-Blockchain Multi-Party Payment Hub". In: *IEEE Transactions on Network and Service Management* 18.4 (2021/12), pp. 4450–4461. ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3101808. URL: https://ieeexplore.ieee.org/abstract/document/9504436.

[Yic20] Pu Yichao. "Verification and Optimization of Metro Fare Clearing Models Based on Travel Route Reconstruction". In: *New Metro* 1.1 (2020/12/02), pp. 34–47. ISSN: 2709-7714. DOI: 10.37819/nm.001.01.0076. URL: https://eaapublishing.org/journals/index.php/nm/article/view/76.

[Yin+24] Zuobin Ying, Qingao Ding, Wenqi Li, Shengmin Xu, and Jinbo Xiong. "Towards Scalable and Secure IoTs Transactions: A New Bi-directional Payment Channel Without Third-Party Monitoring". In: *Information Security and Privacy*. Ed. by Tianqing Zhu and Yannan Li. Singapore: Springer Nature, 2024, pp. 120–139. ISBN: 978-981-97-5101-3. DOI: 10.1007/978-981-97-5101-3_7.

[Yu+18] R. Yu, G. Xue, V. T. Kilari, D. Yang, and J. Tang. "CoinExpress: A Fast Payment Routing Mechanism in Blockchain-Based Payment Channel Networks". In: *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. 2018 27th International Conference on Computer Communication and Networks (ICCCN). 2018/07, pp. 1–9. DOI: 10.1109/ICCCN.2018.8487351.

[Yu+19] Bin Yu, Shabnam Kasra Kermanshahi, Amin Sakzad, and Surya Nepal. "Chameleon Hash Time-Lock Contract for Privacy Preserving Payment Channel Networks". In: *Provable Security*. Ed. by Ron Steinfeld and Tsz Hon Yuen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 303–318. ISBN: 978-3-030-31919-9. DOI: 10.1007/978-3-030-31919-9_18.

[Zab+22] Philipp Zabka, Klaus-T Foerster, Christian Decker, and Stefan Schmid. "Short Paper: A Centrality Analysis of the Lightning Network". In: *Financial Cryptography and Data Security*. Financial Cryptography and Data Security. 2022, p. 11. URL: https://fc22.ifca.ai/preproceedings/39.pdf.

[Zab+24] Philipp Zabka, Klaus-T. Förster, Christian Decker, and Stefan Schmid. "A Centrality Analysis of the Lightning Network". In: *Telecommunications Policy* 48.2 (2024/03/01), p. 102696. ISSN: 0308-5961. DOI: 10.1016/j.telpol.2023.102696. URL: https://www.sciencedirect.com/science/article/pii/S0308596123002070.

[Zha+23] Yi Zhang, Xiaofeng Jia, Bianjing Pan, Jun Shao, Liming Fang, Rongxing Lu, and Guiyi Wei. "Anonymous Multi-Hop Payment for Payment Channel Networks". In: *IEEE Transactions on Dependable and Secure Computing* (2023), pp. 1–11. ISSN: 1941-0018. DOI: 10.1109/TDSC.2023.3262681.

[Ziv+21] Maya Ziv, Liz Izhikevich, Kimberly Ruth, Katherine Izhikevich, and Zakir Durumeric. "ASdb: A System for Classifying Owners of Autonomous Systems". In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021/11/02, pp. 703–719. ISBN: 978-1-4503-9129-0. DOI: 10.1145/3487552.3487853. URL: https://doi.org/10.1145/3487552.3487853.

[ZSQ21] Xiaoxue Zhang, Shouqian Shi, and Chen Qian. *WebFlow: Scalable and Decentralized Routing for Payment Channel Networks with High Resource Utilization*. arXiv, 2021/09/23. DOI: 10.48550/arXiv.2109.11665. arXiv: 2109.11665 [cs]. URL: http://arxiv.org/abs/2109.11665.

[ZSX15] Feng Zhou, Jun-gang Shi, and Rui-hua Xu. "Estimation Method of Path-Selecting Proportion for Urban Rail Transit Based on AFC Data". In: *Mathematical Problems in Engineering* 2015 (2015/09/07). ISSN: 1024-123X. DOI: 10.1155/2015/350397. URL: https://www.hindawi.com/journals/mpe/2015/350397/.

[ZT26] Yuhao Zhou and Stavros Tripakis. *Towards Language Model Guided TLA+ Proof Automation*. arXiv, 2026/02/28. DOI: 10.48550/arXiv.2512.09758. arXiv: 2512.09758 [cs]. URL: http://arxiv.org/abs/2512.09758.

[Zum23] Martin Zumsande. "Bitcoin Core - P2P In Practice". 2023/10/27. URL: https://www.youtube.com/watch?v=dWCPeiIzVco.

[ZYX19] Y. Zhang, D. Yang, and G. Xue. "CheaPay: An Optimal Algorithm for Fee Minimization in Blockchain-Based Payment Channel Networks". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. ICC 2019 - 2019 IEEE International Conference on Communications (ICC). 2019/05, pp. 1–6. DOI: 10.1109/ICC.2019.8761804.

[ZZS21] Z. Zhao, L. Zhou, and C. Su. "Systematic Research on Technology and Challenges of Lightning Network". In: *2021 IEEE Conference on Dependable and Secure Computing (DSC)*. 2021 IEEE Conference on Dependable and Secure Computing (DSC). 2021/01, pp. 1–8. DOI: 10.1109/DSC49826.2021.9346275.

Webpages

[Bar24] Vilius Barbaravičius. *Year-over-Year Data Shows Rising Lightning Network Adoption | CoinGate*. Best Bitcoin & Crypto Payment Processor. 2024/08/13. URL: https://coingate.com/blog/post/lightning-network-year-over-year-data (visited on 2025/01/14).

[Blo] Blockchain.com. *Average Confirmation Time in Bitcoin*. URL: https://www.blockchain.com/explorer/charts/%5Bid%5D (visited on 2026/04/24).

[Blo21] Blockstream Research. *Multi-Hop Locks from Scriptless Scripts*. GitHub. 2021/11/23. URL: https://github.com/BlockstreamResearch/scriptless-scripts/blob/master/md/multi-hop-locks.md (visited on 2026/04/27).

[Col15] Jeff Coleman. *State Channels - an Explanation*. 2015/11/06. URL: https://www.jeffcoleman.ca/state-channels/ (visited on 2025/07/10).

[Cym] Team Cymru. *IP to ASN Mapping Service | Team Cymru*. URL: https://www.team-cymru.com/ip-asn-mapping (visited on 2026/03/03).

[I2P] I2P. *I2P - The Invisible Internet Project | EN*. URL: https://i2p.net/en/ (visited on 2026/03/02).

[Lab] Brainbot Labs. *Raiden Network*. URL: https://raiden.network/ (visited on 2025/08/28).

[Lig] Lightning Labs. *LND Private Altruist Watchtowers*. GitHub. URL: https://github.com/lightninglabs/docs.lightning.engineering/blob/master/docs/lnd/watchtower.md (visited on 2026/03/05).

[Lig26a] Lightning Developers. *BOLT #2: Peer Protocol for Channel Management*. GitHub. 2026. URL: https://github.com/lightning/bolts/blob/master/02-peer-protocol.md (visited on 2026/03/05).

[Lig26b] Lightning Developers. *BOLT #3: Bitcoin Transaction and Script Formats*. GitHub. 2026. URL: https://github.com/lightning/bolts/blob/master/03-transactions.md (visited on 2026/03/05).

[Lig26c] Lightning Developers. *Lightning Network In-Progress Specifications*. GitHub. 2026. URL: https://github.com/lightning/bolts (visited on 2025/08/29).

[Luk] Luke-Jr. *Bitcoin Node Count History*. URL: https://luke.dashjr.org/programs/bitcoin/files/charts/historical.html (visited on 2026/03/03).

[Meh+24] Dhruv Mehta, Tim Ruffing, Jonas Schnelli, and Pieter Wuille. *BIP 324 Version 2 P2P Encrypted Transport Protocol*. GitHub. 2024/07/10. URL: https://github.com/bitcoin/bips/blob/master/bip-0324.mediawiki (visited on 2026/03/05).

[NGH] Till Neudecker, Matthias Grundmann, and Hannes Hartenstein. *Bitcoin Network Monitor - DSN Research Group, KASTEL @ KIT*. URL: https://www.dsn.kastel.kit.edu/bitcoin/ (visited on 2026/03/02).

[Tea24a] Bitcoin Core Security Team. *CVE-2024-52919 - Remote Crash Due to Addr Message Spam*. Bitcoin Core. 2024/07/31. URL: https://bitcoincore.org/en/2024/07/31/disclose-addrman-int-overflow/ (visited on 2026/03/03).

[Yeo] Addy Yeow. *Bitnodes*. URL: https://bitnodes.io/ (visited on 2026/03/02).

[Ygg] Yggdrasil. *Yggdrasil Network*. Yggdrasil Network. URL: https://yggdrasil-network.github.io/ (visited on 2026/03/02).

A. Appendix

A.1. Formalization of Messages in Lightning Specification

Tables A.1 to A.13 show messages of the Lightning specification with their fields. The tables show how we model each field in the TLA$^+$ specification of Lightning.

A.2. Variables of *SpecificationI*

The module *SpecificationI* uses several variables to represent the state of the system. A subset of these variables is used by each of the other modules. We give an overview of these variables by describing the purpose of each variable used in *SpecificationI*. There are four groups of variables:

- *Global variables*
  - The variable *LedgerTx* models the blockchain. The variable is a set of transactions that have been published on the blockchain. The modeling of transactions is described in Section 3.3.2.5.
  - The variable *LedgerTime* is an integer that models the current height of the blockchain which is used in Lightning as a logical time. The modeling of time is presented in Section 3.3.5.1.
  - The variable *TxAge* is a mapping of transaction ids to integers. For each transaction that is contained in *LedgerTx*, the variable *TxAge* maps the transaction's id to the age of the transaction. The age of a transaction is defined as the difference between the current time and the publication time of the transaction.
  - The variable *Messages* is a set of messages that are in-transit, i.e. a user has sent the message but the receiver has not yet received the message. This variable is used only for messages between the sender and the receiver of a multi-hop payment to initiate the payment. How we model the exchange of messages is described in Section 3.3.2.1.
- *User variables* are variables that exist once for each modeled user of the payment channel network. We model such a set of one variable per user with a function that assigns to each user identifier the respective variable. E.g., the variable *UserPayments*

Table A.1.: Fields of 'open_channel' message (Part 1/2).

Variable	Description	Formalization
chain_hash	Hash of the blockchain underlying the payment channel.	Not required because formalization considers only one blockchain.
temporary_channel_id	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
funding_satoshis	Number of satoshis that the funder deposits into the channel.	Field 'Capacity'
push_msat	Number of millisatoshi that the funder is giving to the other party.	Not included for simplicity.
dust_limit_satoshis	No output is created with an amount less than this value. This prevents outputs from being created that cannot be spent economically because the amount of fees required to spend the output would be higher than the value of the output.	Not required because fees are not modeled.
max_htlc_value_in_flight_msat	Maximum amount of millisatoshi that can be part of HTLCs.	Not included for simplicity.
channel_reserve_satoshis	Amount of satoshis that both users need to keep as their balance and cannot spend to disincentivize cheating attempts.	Not required because the formalization does not consider the game theoretic aspect of whether a user might want to cheat but instead shows that cheating never succeeds.

is a function and *UserPayments[u]* for a user u is the variable that contains payment records of user u. The names of all user variables start with the prefix *User*.

- The variables *UserPayments[u]* are sets that contain payment records for user u. Each of these payment records has the following fields: *amount*, *sender*, *receiver*, *state*, and *id*. The field *state* is initially set to `new` and can either change to

Table A.2.: Fields of 'open_channel' message (Part 2/2).

Variable	Description	Formalization
htlc_minimum_msat	Amount that any HTLC must at least have.	Not required for security properties.
feerate_per_kw	Fee rate for commitment and HTLC transactions.	Not required as blockchain fees are not modeled.
to_self_delay	Number of blocks that an output of the counterparty must be locked until the counterparty can spend it.	Modeled as constant 'TO_SELF_DELAY'.
max_accepted_htlcs	Maximum number of HTLCs that are accepted at the same time to ensure that messages do not grow too large.	Not required.
funding_pubkey	Public key used for locking the funding output that can only be spent using signatures of keys of both users.	Modeled as the user's public key.
revocation_basepoint	Point used to calculate revocation keys.	Modeled in form of a public/private revocation key pair.
payment_basepoint	Point used to calculate the payment keys.	Modeled as the user's public key.
delayed_payment_basepoint	Point used to calculate the delayed payment keys.	Modeled as the user's public key.
htlc_basepoint	Point used to calculate the HTLC keys.	Modeled as the user's public key.
first_per_commitment_point	Point to calculate keys for the first commitment transaction.	Only revocation key is rotated. Modeled as key index variable as part of the key.
channel_flags	Flags, e.g. whether this channel is to announced publicly to the P2P network.	Not required.

Table A.3.: Fields of 'accept_channel' message (Part 1/2). With the exception of minimum_depth all fields are also described in Tables A.1 and A.2.

Variable	Description	Formalization
temporary_channel_id	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
dust_limit_satoshis	No output is created with an amount less than this value. This prevents outputs from being created that cannot be spent economically because the amount of fees required to spend the output would be higher than the value of the output.	Not required because fees are not modeled.
max_htlc_value_in_flight_msat	Maximum amount of millisatoshi that can be part of HTLCs.	Not included for simplicity.
channel_reserve_satoshis	Amount of satoshis that both users need to keep as their balance and cannot spend to disincentivize cheating attempts.	Not required because the formalization does not consider the game theoretic aspect of whether a user might want to cheat but instead shows that cheating never succeeds.

`aborted` or `processed`. The variables *UserPayments[u]* are initialized with sets of payment records for payments that should be sent or received by user u.

- The variables *UserExtBalance[u]* are integers that model the external balance of each user u. The external balance is the balance that a user has on the blockchain.
- The variables *UserChannelBalances[u]* is a mapping of a user's channels to integers to model the balance that user u has in each payment channel. In contrast to *ChannelUserBalance[c][u]*, this balance changes only when a channel is opened or closed or when a payment has been processed. The purpose of the variable is that the sum of all balances in all channels corresponds to the total balance that a user has in all of the user's payment channels which is a variable used in the security property.

Table A.4.: Fields of 'accept_channel' message (Part 2/2).

Variable	Description	Formalization
htlc_minimum_msat	Amount that any HTLC must at least have.	Not required for security properties.
minimum_depth	Number of blocks created after a transaction has been published until the transaction is considered confirmed.	Not required as transactions are modeled as being finally confirmed immediately.
to_self_delay	Number of blocks that an output of the counterparty must be locked until the counterparty can spend it.	Modeled as constant 'TO_SELF_DELAY'.
max_accepted_htlcs	Maximum number of HTLCs that are accepted at the same time to ensure that messages do not grow too large.	Not required because message sizes are theoretically not limited.
funding_pubkey	Public key used for locking the funding output that can only be spent using signatures of keys of both users.	Modeled as the user's public key.
revocation_basepoint	Point used to calculate revocation keys.	Modeled in form of a public/private revocation key pair.
payment_basepoint	Point used to calculate the payment keys.	Modeled as the user's public key.
delayed_payment_basepoint	Point used to calculate the delayed payment keys.	Modeled as the user's public key.
htlc_basepoint	Point used to calculate the HTLC keys.	Modeled as the user's public key.
first_per_commitment_point	Point to calculate keys for the first commitment transaction.	Only revocation key is rotated. Modeled as key index variable as part of the key.

Table A.5.: Fields of 'funding_created' message.

Variable	Description	Formalization
temporary_channel_id	Temporary id to describe the channel.	Not required because channel is uniquely identified by the pair of the channel parties.
funding_txid	ID of the funding transaction	.
funding_output_index	Output ID for the funding output of the funding transaction.	Not required as funding transaction contains only one output.
signature	Signature of the first commitment transaction	First commitment transaction signed by inserting private key.

Table A.6.: Fields of 'funding_signed' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
signature	Signature of the first commitment transaction	First commitment transaction signed by inserting private key.

- The variables *UserHonest[u]* are boolean variables that define whether user u is honest. An honest user acts compliant to the Lightning protocol. All variables *UserHonest[u]* are initialized to either TRUE or FALSE and do not change.
- The variables *UserNewPayments[u]* are sets that contain records for payments that are queued to be sent by the user. In contrast, *UserPayments[u]* contains all of a user's sent and received payments.
- The variables *UserPreimageInventory[u]* are sets of preimages that are known by user u.

Table A.7.: Fields of 'channel_ready' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
second_per_commitment_point	Point to calculate keys for the second commitment transaction	Only revocation key is rotated. Modeled as key index variable as part of the key.

Table A.8.: Fields of 'commitment_signed' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
signature	Signature of the new commitment transaction	New commitment transaction signed by inserting private key.
num_htlcs	Number of HTLCs in the new commitment transaction	Not required because the field is redundant.
num_htlcs * signature	Signature of each HTLC transaction	HTLC transaction signed by inserting private key.

Table A.9.: Fields of 'revoke_and_ack' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
per_commitment_secret	Secret for key derivation of keys for the previous commitment transaction.	Only revocation key is rotated. Modeled as sending the private revocation key.
next_per_commitment_point	Point to calculate keys for the new commitment transaction	Only revocation key is rotated. Modeled by sending public revocation key.

  - The variables *UserPaymentSecretForPreimage[u]* are functions that map a payment secret to a preimage. These functions contain an entry for each payment that is received by user u if user u has already generated a preimage for the payment.

- *Channel variables* exist once for each modeled channel of the payment channel network. Analogously to user variables, channel variables are also modeled with a function that assigns the respective variable to each channel identifier.
  - The variables *ChannelMessages[c]* are sequences of messages that have been sent between the two users of channel c. When a user sends a message in channel c, the message is appended to *ChannelMessages[c]*. Each user can receive the first message in *ChannelMessages[c]* for which the user u is the receiver. Our model of messages is explained in Section 3.3.2.1.
  - The variables *ChannelUsedTransactionIds[c]* are sets of transaction ids that have been used inside channel c. Each channel is assigned a unique range of natural numbers from which the transaction ids in this channel are drawn. Our model of transaction ids is presented in Section 3.3.2.5.

Table A.10.: Fields of a Lightning 'invoice'.

Variable	Description	Formalization
timestamp	Unix Timestamp	Not required.
p	Payment hash	'hash' field of invoice
s	Payment secret	'paymentSecret' field of invoice
d	Description of purpose of payment	Not required.
m	Additional metadata	Not required.
n	Public key of the payment's receiver.	Not required.
h	Description of purpose of payment	Not required.
x	Expiry time	Not required.
c	Minimal delta between HTLC time-locks for last HTLC	Not required because formalized as a constant default value.
f	Fallback on-chain Bitcoin address	Not required.
r	Routing information	Not required.
9	Feature bits	Not required.
signature	Signature of above fields	Not included. We instead assume that the receiver can verify the integrity of the invoice.

Table A.11.: Fields of 'update_add_htlc' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	'id' field of HTLC
amount_msat	Amount of HTLC in millisatoshi	'amount' field of HTLC
payment_hash	Hash of HTLC	'hash' field of HTLC
cltv_expiry	Timelock of HTLC	'absTimelock' field of HTLC
onion_routing_packet	Data for forwarding to next hop including encrypted payload for next hop.	'dataForNextHop' field of HTLC

Table A.12.: Fields of 'update_fulfill_htlc' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	Not required.
payment_preimage	Preimage for HTLC	'preimage' field

Table A.13.: Fields of 'update_fail_htlc' message.

Variable	Description	Formalization
channel_id	ID of the channel derived from funding transaction	Not required.
id	ID of the HTLC	'id' field of HTLC
len	Length of the reason field	Not required.
reason	Reason why HTLC failed	Not required.

  - The variables *ChannelPendingBalance[c]* contain integers that represent the pending balance of channel c. By pending balance, we refer to the amount of funds that is part of a payment that is in process and it is undecided yet whether the payment will fail or succeed.

- ⟨*Channel and user*⟩ *variables* represent a user's state that is specific to a channel. These variables are modeled with a function that assigns the respective variable to a given channel identifier and a given user identifier.
  - The variables *ChannelUserBalance[c][u]* are integers that represent the current balance of user u in channel c. These variables contain only the balance that user u is guaranteed to receive when the channel c is closed. In particular, it does not include the amounts of payments that are currently being processed and it is not clear whether the payment will be successful or fail.
  - The variables *ChannelUserState[c][u]* are strings that identify the current protocol state of user u in channel c.
  - The variables *ChannelUserVars[c][u]* are records that contain several variables of user u in channel c, e.g., sets of incoming and outgoing HTLCs.
  - The variables *ChannelUserDetailVars[c][u]* are records that contain several low-level variables of user u in channel c, e.g., keys and ids of commitment transactions.
  - The variables *ChannelUserInventory[c][u]* are records that contain a set of the (funding and commitment) transactions for channel c known by user u and a set of keys for channel c know by user u.

A.3. Proofs

This section contains various proofs for lemmas that we introduced in the previous chapters. Most proofs are written as structured proofs following the approach presented in [Lam12].

A.3.1. Proof of Eq. (4.1)

We recall Eq. (4.1) which was defined on page 108:

Consider a state t that is in a zone Z that contains states whose clock values of the clock x differ.

$$\neg(t \rightsquigarrow T^x_{t.x-1}(t)) \Leftrightarrow t \text{ is a zone representative}$$

1. $\neg(t \rightsquigarrow T^x_{t.x-1}(t)) \Rightarrow t$ is a zone representative
   1.1. Suffices Assume: t is not a zone representative
   Prove: $t \rightsquigarrow T^x_{t.x-1}(t)$
   Proof: By inverting the implication in 1
   1.2. $\exists u \in Z : u.x < t.x$
   Proof: By definition of a zone representative (see Definition 9), the state t is a zone representative if it is the state with the lowest value of clock x in the zone. Given that the state t is not a zone representative (see 1.1), by definition of the zone representative in Definition 9, there must be a state u in the zone that has a lower clock value for clock x than the state t.
   1.3. $\exists u \in Z : u.x = t.x - 1$
   Proof: From the third condition of Definition 9, it follows from 1.2 that there must be a state u in the zone whose clock value for clock x equals $t.x - 1$.
   1.4. $T^x_{t.x-1}(t) \in Z$
   Proof: Because the state u in 1.3 is in the zone, it follows from the first two conditions of Definition 9 that the values of all variables and all clocks except clock x in state u are equal to the respective values in state t, i.e., the states u and t differ only in the clock value of the clock x. Thus, the state u in 1.3 can be written as $T^x_{t.x-1}(t)$.
   1.5. Q.E.D.
   Proof: By the fourth condition of Definition 9, it follows from 1.4 that the state t is time-abstracting simulated by the state $T^x_{t.x-1}(t)$.
2. $\neg(t \rightsquigarrow T^x_{t.x-1}(t)) \Leftarrow t$ is a zone representative
   2.1. Suffices Assume: $t \rightsquigarrow T^x_{t.x-1}(t)$
   Prove: t is not a zone representative
   Proof: By inverting the implication in 2
   2.2. $T^x_{t.x-1}(t) \in Z$
   Proof: Given that the state t is time-abstracting simulated by the state $T^x_{t.x-1}(t)$ (see 2.1), the state $T^x_{t.x-1}(t)$ must be in zone Z by the fifth condition of Definition 9 because the state $T^x_{t.x-1}(t)$ is, by definition, varequal to state t and the values of all clocks except clock x are the same between states t and $T^x_{t.x-1}(t)$.
   2.3. Q.E.D.
   Proof: By definition of the zone representative in Definition 9, it follows from 2.2 that t is not a zone representative because $T^x_{t.x-1}(t).x < t.x$.
3. Q.E.D.

Proof: By 1 and 2

A.3.2. Proof that Eq. (4.2) is Equivalent to Definition 11

Consider a real-time system with state space Σ and a set of clocks $\mathcal{X}$. Consider a state $t \in \Sigma$ that is in a zone $Z \subseteq \Sigma$ that contains states whose clock values of the clock $x \in \mathcal{X}$ differ. We first show that

$$\neg(t \rightsquigarrow T^x_{t.x-1}(t)) \Leftrightarrow \exists t_1 \in \Sigma : t \xrightarrow{([Next_S]_{v_S})^+} t_1 \wedge \exists u \in \Sigma : t_1 \xrightarrow{Next_S} u \wedge \neg(T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} T^x_{u.x-1}(u)) \tag{A.1}$$

1. Assume: $\neg(t \rightsquigarrow T^x_{t.x-1}(t))$
   Prove: $\exists t_1 \in \Sigma : t \xrightarrow{([Next_S]_{v_S})^+} t_1 \wedge \exists u \in \Sigma : t_1 \xrightarrow{Next_S} u \wedge \neg(T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} T^x_{u.x-1}(u))$
2. Choose $t_1 \in \Sigma, t_2 \in \Sigma$ so that $t \xrightarrow{([Next_S]_{v_S})^+} t_1 \wedge T^x_{t.x-1}(t) \xrightarrow{([Next_S]_{v_S})^+} t_2$ so that both states t_1 and t_2 are reached by the same sequence of actions and $\neg(t_1 \rightsquigarrow t_2)$ and $\exists u \in \Sigma : t_1 \xrightarrow{Next_S} u$ and $\neg(\exists v \in \Sigma : t_2 \xrightarrow{Next_S} v \wedge u \rightsquigarrow v)$
   Proof: We write $s \xrightarrow{A^+} t$ if state t can be reached from state s by a sequence of one or more A steps. The states t_1 and t_2 exist with these conditions by definition of time-abstracting simulation (Definition 5) because $\neg(t \rightsquigarrow T^x_{t.x-1}(t))$ (see 1).
3. There is no step of $AdvanceTime_S$ in the sequence of actions between the states t and t_1
   Proof: If there was an $AdvanceTime_S$ step between the states t and t_1, then the first $AdvanceTime_S$ step would have increased the clock value of clock x from $t.x$ to a value $d > t.x$. This step would have been matched in the series of states between $T^x_{t.x-1}(t)$ and t_2 by an $AdvanceTime_S$ step that increases the time from $t.x - 1$ to d because of the following reasons. The $AdvanceTime_S$ step between the states t and t_1 was matched by an $AdvanceTime_S$ step because the steps between the states $T^x_{t.x-1}(t)$ and t_2 have the same sequence of actions as the steps between the states t and t_1 (by 2). The $AdvanceTime_S$ step between the states $T^x_{t.x-1}(t)$ and t_2 can increase the clock value of the clock x to d because, if it was not possible to increase from $t.x - 1$ to $t.x$ then an $AdvanceTime_S$ step would not be possible at all, and, if a step from $t.x - 1$ to $t.x$ and from $t.x$ to d is possible then it is also possible from $t.x - 1$ to d. Thus, the $AdvanceTime_S$ step between the states t and t_1 and the $AdvanceTime_S$ step between the states $T^x_{t.x-1}(t)$ and t_2 would arrive at the same state and, consequently, it would follow that $t_1 \rightsquigarrow t_2$ which conflicts with 2. Thus, there is no $AdvanceTime_S$ step between the states t and t_1.
4. $t_2 = T^x_{t_1.x-1}(t_1)$
   Proof: By 3, the clock value of the clock x does not change between the states t and t_1 and between the states $T^x_{t.x-1}(t)$ and t_2.

5. $\exists u \in \Sigma : t_1 \xrightarrow{Next_S} u \wedge \neg(\exists v \in \Sigma : T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} v \wedge u \rightsquigarrow v)$

   Proof: By 2

6. Choose $u \in \Sigma$ so that $t_1 \xrightarrow{Next_S} u \wedge \neg(\exists v \in \Sigma : T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} v \wedge u \rightsquigarrow v)$

   Proof: By 5, such a state u exists.

7. $\neg(\, T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} T^x_{u.x-1}(u) \wedge u \rightsquigarrow T^x_{u.x-1}(u))$

   7.1. Case: $t_1 \xrightarrow{InnerNext_S} u$

   Proof: In an $InnerNext_S$ step, the values of the clocks do not change. Thus, a step from state t_1 to state u must be matched by a step from $t_2 = T^x_{t_1.x-1}(t_1)$ to $T^x_{u.x-1}(u)$ which implies that v in 6 can only be $T^x_{u.x-1}(u)$.

   7.2. Case: $t_1 \xrightarrow{AdvanceTime_S} u$

   Proof: If the value of the clock x does not change in the step from state t_1 to state u then see 7.1. Otherwise, the value of the clock x is increased in the step from state t_1 to state u from $t_1.x$ to some value d. An $AdvanceTime_S$ step that increases the clock x starting from state $t_2 = T^x_{t_1.x-1}(t_1)$ from $t_1.x - 1$ to d is not possible. If such a step is not possible then increasing the clock x from $t_1.x - 1$ to $d - 1$ is also not possible because, if this was possible and it is possible to increase the clock value of the clock x from $t_1.x$ to d, then it must also be possible to increase the clock value of the clock x from $t_1.x - 1$ to d.

8. Q.E.D.

   Proof: By 1, 2, 6, and 7

We continue to show that the definition of zone time points in Eq. (4.2) is equivalent to the definition of zone time points in Definition 11. The set of zone time points for a state s and a clock x is defined in Eq. (4.2) as follows:

$$\begin{aligned}\{d \in \mathbb{N} \mid \exists t \in \Sigma : s &\xrightarrow{([Next_S]_{v_S})^+} t \\ &\wedge\ t.x = d \\ &\wedge\ \neg(t \rightsquigarrow T^x_{t.x-1}(t)) \\ \}&\end{aligned} \tag{A.2}$$

Using the equivalence shown in Eq. (A.1), we can write this definition as follows:

$$\begin{aligned}\{d \in \mathbb{N} \mid \exists t \in \Sigma \ : \ & s \xrightarrow{([Next_S]_{v_S})^+} t \\ & \wedge\ t.x = d \\ & \wedge\ \exists t_1 \in \Sigma : t \xrightarrow{([Next_S]_{v_S})^+} t_1 \\ & \qquad \wedge\ \exists u \in \Sigma : t_1 \xrightarrow{Next_S} u \wedge \neg(\, T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} T^x_{u.x-1}(u)) \\ \}\end{aligned} \tag{A.3}$$

In step 3 of the proof of Eq. (A.1), we have shown that there is no step of $AdvanceTime_S$ between the states t and t_1. Thus, the clock value of the clock x in state t_1 equals the respective clock value in state t which equals d, i.e., $d = t.x = t_1.x$. Therefore, the state t is not needed in the definition and the two series of steps from s to t and from t to t_1 can be merged to $s \xrightarrow{([Next_S]_{v_S})^+} t_1$. Thereby, we can write the definition as follows:

$$\begin{aligned}\{d \in \mathbb{N} \mid \exists t \in \Sigma : \ & s \xrightarrow{([Next_S]_{v_S})^+} t_1 \\ & \wedge\ t_1.x = d \\ & \wedge\ \exists u \in \Sigma : t_1 \xrightarrow{Next_S} u \wedge \neg(\, T^x_{t_1.x-1}(t_1) \xrightarrow{Next_S} T^x_{u.x-1}(u)) \\ \}\end{aligned} \tag{A.4}$$

After renaming the state t_1 to t, we obtain Definition 11 from Eq. (A.4).

A.3.3. Proof of Lemma 2

We recall Lemma 2 which was defined on page 116:

Lemma 2. *From $Assumption$ 4.3, it follows that $Inv_{S'}$ is an invariant of specification $Spec_{S'}$:*

$$Spec_{S'} \Rightarrow \Box Inv_{S'}$$

And we recall the definition of $Inv_{S'}$ (see page 115):

Definition 18 ($Inv_{S'}$)**.** Define an invariant $Inv_{S'}$ of a state s of specification $Spec_{S'}$ as:

$$Inv_{S'}(s) \equiv \forall x \in \mathcal{X} : s.mappedClock^x \geq max(\{n \in relZTP^x(\ddot{s}) \mid n \leq s.x\})$$

1. $Init_{S'} \Rightarrow Inv_{S'}$

   By definition of $Init_{S'}$, it holds that $\forall x \in \mathcal{X} : s.mappedClock^x = s.x$. The statement follows because $\forall x \in \mathcal{X} : max(\{n \in relZTP^x(\ddot{s}) \mid n \leq s.x\}) \leq s.x$.
2. $Inv_{S'} \wedge [Next_{S'}]_{v_{S'}} \Rightarrow Inv'_{S'}$

We refer to the step's starting state as s and to the ending state as t.

2.1. $Inv_{S'} \wedge (v'_{S'} = v_{S'}) \Rightarrow Inv'_{S'}$

Proof: Assume that $Inv_{S'}$ and $v'_{S'} = v_{S'}$. It directly follows that $Inv'_{S'}$.

2.2. $Inv_{S'} \wedge AdvanceTime_{S'} \Rightarrow Inv'_{S'}$

2.2.1. Assume: $\forall x \in \mathcal{X} : s.mappedClock^x \geq max(\{n \in relZTP^x(\ddot{s}) \mid n \leq s.x\}) \wedge AdvanceTime_{S'}$

Prove: $\forall x \in \mathcal{X} : t.mappedClock^x \geq max(\{n \in relZTP^x(\ddot{t}) \mid n \leq t.x\})$

2.2.2. $\forall x \in \mathcal{X} : t.mappedClock^x = max(\{s.mappedClock^x\} \cup \{n \in relZTP^x(\ddot{s}) \mid n \leq t.x\})$

Proof: By definition of $AdvanceTime_{S'}$

2.2.3. $\forall x \in \mathcal{X} : t.mappedClock^x \geq max(\{n \in relZTP^x(\ddot{s}) \mid n \leq t.x\})$

Proof: By 2.2.2

2.2.4. $\forall x \in \mathcal{X} : max(\{n \in relZTP^x(\ddot{s}) \mid n \leq t.x\}) \geq max(\{n \in relZTP^x(\ddot{t}) \mid n \leq t.x\})$

Proof: Because, by Assumption 4.3, $relZTP^x(\ddot{t}) \subseteq relZTP^x(\ddot{s})$

2.2.5. Q.E.D.

Proof: By 2.2.4

2.3. $Inv_{S'} \wedge InnerNext_{S'} \Rightarrow Inv'_{S'}$

Proof: Let $x \in \mathcal{X}$ be arbitrary but fixed.

2.3.1. Assume: $Inv_{S'} \wedge InnerNext_{S'}$

Prove: $Inv'_{S'}$

2.3.2. Case: $x' = x$

2.3.2.1. $s.mappedClock^x = t.mappedClock^x$

Proof: By 2.3.2 and the definition of $InnerNext_{S'}$

2.3.2.2. $s.x = t.x$

Proof: By 2.3.2

2.3.2.3. $relZTP^x(\ddot{t}) \subseteq relZTP^x(\ddot{s})$

Proof: By Assumption 4.3

2.3.2.4. $max(\{n \in relZTP^x(\ddot{t}) \mid n \leq t.x\}) \leq max(\{n \in relZTP^x(\ddot{s}) \mid n \leq s\})$

Proof: By 2.3.2.2 and 2.3.2.3

2.3.2.5. Q.E.D.

Proof: By 2.3.2.4, $Inv_{S'}$, and 2.3.2.1

2.3.3. Case: $x' \neq x$

2.3.3.1. $t.mappedClock^x = t.x$

Proof: By 2.3.3 and the definition of $InnerNext_{S'}$

2.3.3.2. Q.E.D.

Proof: By 2.3.3.1 and because $max(\{n \in relZTP^x(\ddot{t}) \mid n \leq t.x\}) \leq t.x$

2.3.4. Q.E.D.

Proof: By 2.3.2 and 2.3.3 because these cover all possible cases.

2.4. Q.E.D.

Proof: Because 2.1, 2.2, 2.3, and $[Next_{S'}]_{v_{S'}} = (AdvanceTime_{S'} \vee InnerNext_{S'}) \vee (v'_{S'} = v_{S'})$.

3. Q.E.D.

By 1, 2, and the definition of $Spec_{S'}$.

A.3.4. Proof of Lemma 3

We recall Lemma 3 which was defined on page 117:

Lemma 3. *From* $Assumptions$ 4.1 *and* 4.2*, it follows that*

$$Spec_{S'} \Rightarrow \Lambda(Spec_{\hat{S}})$$

For the proof, we define $\Lambda_i(F)$ analogously to $\Lambda(F)$ (see Definition 17 on page 115) with the difference that $\Lambda_i(F)$ maps only a finite number $i \in \mathbb{N}$ of clocks:

Definition 22 ($\Lambda_i(F)$)**.**

$$\begin{aligned}\Lambda_i(F) = F \text{ WITH } x_1 &\leftarrow mappedClock^{x_1},\\ x_2 &\leftarrow mappedClock^{x_2},\\ x_3 &\leftarrow mappedClock^{x_3},\\ &\ldots\\ x_i &\leftarrow mappedClock^{x_i}\end{aligned}$$

Proof: In the proof, we use the shorthand $L_{S'}$ for $Liveness_{S'}$ and $L_{\hat{S}}$ for $Liveness_{\hat{S}}$.

1. $Spec_{S'} \Rightarrow \Lambda(Init_{S'})$

   Proof: Recall that, by definition, $Init_{S'} = Init_S \wedge \forall x \in \mathcal{X} : mappedClock^x = x$ and $\Lambda(Init_{\hat{S}}) = Init_{\hat{S}}$ WITH $x_1 \leftarrow mappedClock^{x_1}, x_2 \leftarrow mappedClock^{x_2}, x_3 \leftarrow mappedClock^{x_3}, \ldots$ and $Init_{\hat{S}} = Init_S$.
   Because $\forall x \in \mathcal{X} : mappedClock^x = x$, it trivially holds that $Init_{S'} \Rightarrow \Lambda(Init_{\hat{S}})$.

2. $Spec_{S'} \Rightarrow \Lambda(\Box[Next_{\hat{S}}]_{v_{\hat{S}}} \wedge L_{\hat{S}})$

   In the following, we refer to the step's starting state as s and to the ending state as t.

   2.1. Suffices Assume: $Inv_{S'} \wedge [Next_{S'}]_{v_{S'}}$
   Prove: $\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$

   Proof: By Lemma 2, it holds that $Spec_{S'} \Rightarrow \Box Inv_{S'}$.
   Thus, if $Inv_{S'} \wedge [Next_{S'}]_{v_{S'}} \Rightarrow \Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$ it holds that $Spec_{S'} \Rightarrow \Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$. It follows that $Spec_{S'} \Rightarrow \Lambda(\Box[Next_{\hat{S}}]_{v_{\hat{S}}} \wedge L_{\hat{S}})$ because $L_{S'} = L_{\hat{S}}$ and $L_{S'}$ is defined as fairness for subactions of $Next_S$.

   2.2. Case: $v_{S'}{}' = v_{S'}$

   Proof: From the assumption $v'_{S'} = v_{S'}$ it follows that $\Lambda(v'_{\hat{S}}) = \Lambda(v_{\hat{S}})$ which implies $\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$.

2.3. Case: $AdvanceTime_{S'} \wedge v'_{S'} \neq v_{S'}$

2.3.1. $\exists x \in \mathcal{X} : t.mappedClock^x > s.mappedClock^x$

Proof: Because $v'_{S'} \neq v_{S'}$ and, by definition of $AdvanceTime_{S'}$ all variables v_S are unchanged.

Define: $\mathcal{Y} \subseteq \mathcal{X}$ to be the set of clocks y for which $t.mappedClock^y > s.mappedClock^y$.
Prove: $\Lambda(AdvanceTime_{\hat{S}})$
We prove that $\Lambda(AdvanceTime_{\hat{S}})$ by showing that the properties of $AdvanceTime_{\hat{S}}$ as defined in Definition 13 are fulfilled.

2.3.2. $\Lambda(\exists x \in \mathcal{X} : t.x > s.x)$

Proof: By 2.3.1 and the definition of Λ.

2.3.3. $\Lambda(\forall x \in \mathcal{X} : s.x = \bot \Rightarrow t.x = \bot)$

Proof: By Definition 16 of $AdvanceTime_{S'}$ and the definition of Λ.

We prove the following statement because we will need it in the next steps.

2.3.4. $\forall x \in \mathcal{X} : s.mappedClock^x \neq \bot \Rightarrow s.x \neq \bot$

Proof: By definition of $AdvanceTime_{S'}$ and because the other subactions of $Next_{S'}$ leave $mappedClock^x$ unchanged.

2.3.5. $\Lambda(\forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow t.x = s.x \vee (t.x \in relZTP^x(s) \wedge t.x \geq s.x))$

2.3.5.1. $\forall x \in \mathcal{X} : t.x \neq \bot \Rightarrow t.mappedClock^x = max(\{s.mappedClock^x\} \cup \{n \in relZTP^x(\ddot{s}) \mid n \leq t.x\})$

Proof: By Definition 16 of $AdvanceTime_{S'}$

2.3.5.2. $\forall x \in \mathcal{X} : t.x \neq \bot \Rightarrow t.mappedClock^x \in (\{s.mappedClock^x\} \cup relZTP^x(\ddot{s})) \wedge t.mappedClock^x \geq s.mappedClock^x$

Proof: Because for any set $S \subseteq \mathbb{N}_0$ it holds that $max(S) \in S$ and for any $s \in S$ it holds that $max(S) \geq s$

2.3.5.3. $\forall x \in \mathcal{X} : \Lambda(s.x \neq \bot) \Rightarrow \Lambda(t.x = s.x) \vee (\Lambda(t.x) \in relZTP^x(\ddot{s}) \wedge \Lambda(t.x \geq s.x))$

Proof: By definition of $\Lambda(F)$ and 2.3.4

2.3.5.4. $relZTP^x(\ddot{s}) \subseteq \Lambda(relZTP^x(s))$

Proof: By Assumption 4.3 because $\ddot{s} = \Lambda(s) \vee \Lambda(s) \xrightarrow{AdvanceTime_S} \ddot{s}$.

2.3.5.5. Q.E.D.

By 2.3.5.3 and 2.3.5.4

2.3.6. $\Lambda(\forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b)$

2.3.6.1. $\forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b$

Proof: By the definitions of $AdvanceTime_{S'}$ and $AdvanceTime_S$

2.3.6.2. $\Lambda(\forall x \in \mathcal{X} : s.x \neq \bot) \Rightarrow \forall b \in B^x(s) : s.x \leq b \Rightarrow t.x \leq b$

Proof: By 2.3.4 and the definition of Λ

2.3.6.3. $\Lambda(\forall x \in \mathcal{X} : s.x \neq \bot \Rightarrow \forall b \in B^x(s)) : s.x \leq b \Rightarrow t.x \leq b$

Proof: By Assumption 4.1, the value of $B^x(s)$ is independent of the value of the clocks in state s and, thus, it holds that $\forall x \in \mathcal{X} : B^x(s) = \Lambda(B^x(s))$

2.3.6.4. Suffices Assume: $\Lambda(x \in \mathcal{X}, s.x \neq \bot, b \in B^x(s), s.x \leq b)$
Prove: $\Lambda(t.x \leq b)$

2.3.6.5. $s.x \leq b$

Proof: We prove this by contradiction. Assume that $b < s.x$. Then $(b+1) \leq s.x$. Because $(b+1) \in relZTP^x(s)$ by Assumption 4.2 and because Inv it holds that $s.mappedClock^x \geq (b+1) > b$. This is a contradiction to $b \geq s.mappedClock^x$ which can be written as $\Lambda(b \geq s.x)$.

2.3.6.6. Q.E.D.

Proof: From 2.3.6.3 it follows that $t.x \leq b$. By the definition of $AdvanceTime_{S'}$ it holds that $t.mappedClock^x \leq t.x$. Thus, it holds that $\Lambda(t.x) = t.mappedClock^x \leq b$.

2.3.7. Q.E.D.

By definition of $AdvanceTime_{\hat{S}}$, the steps 2.3.2, 2.3.3, 2.3.5, and 2.3.6 imply that $\Lambda(AdvanceTime_{\hat{S}})$ which implies that $\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$.

2.4. Case: $InnerNext_{S'} \wedge v'_{S'} \neq v_{S'}$

Proof: Informally speaking, we show that every $InnerNext$ step that is allowed in state s by specification S' is also allowed in state $\ddot{s}$ by specification $\hat{S}$. Proof idea: Proof by contradiction. But first handle the special case, that $\Lambda(\ddot{s}) = \ddot{s}$:

2.4.1. Case: $\Lambda(\ddot{s}) = \ddot{s}$

2.4.1.1. $InnerNext_S$

Because, by 2.4, it holds that $InnerNext_{S'}$

2.4.1.2. $InnerNext_{\hat{S}}$

Because $InnerNext_S = InnerNext_{\hat{S}}$

2.4.1.3. $Next_{\hat{S}}$

By definition of $Next_{\hat{S}}$ it holds that $InnerNext_{\hat{S}} \Rightarrow Next_{\hat{S}}$

2.4.1.4. $\Lambda(\ddot{t}) = \ddot{t}$

Because $\Lambda(\ddot{s}) = \ddot{s}$ and, by definition of $InnerNext_{S'}$, the values of the variables $mappedClock^x$ are unchanged and, by definition of $InnerNext_{S'}$ and $InnerNext_S$, the values of the clocks are unchanged.

2.4.1.5. Q.E.D.

Because $Next_{\hat{S}}$ and $\Lambda(\ddot{s}) = \ddot{s}$ and $\Lambda(\ddot{t}) = \ddot{t}$ it holds that $\Lambda(Next_{\hat{S}})$

2.4.2. Case: $\Lambda(\ddot{s}) \neq \ddot{s}$

2.4.2.1. $\exists x \in \mathcal{X} : s.mappedClock^x \neq s.x$

Proof: By 2.4.2.

2.4.2.2. Suffices Assume: $\neg\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$
Prove: FALSE

Because FALSE implies that the assumption is wrong.

Define: $\chi \in \mathbb{N}$ so that $\chi = 1 + min(\{n \in \mathbb{N} : \Lambda_n(Next_S) \wedge \neg\Lambda_{n+1}(Next_S)\})$

By assumption in 2.4.2.2, it holds that $\neg\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$ which implies that $\neg\Lambda(InnerNext_{\hat{S}})$ and $\neg\Lambda(v_{\hat{S}}{}' = v_{\hat{S}})$.

Because $InnerNext_S = InnerNext_{\hat{S}}$, it follows that $\neg\Lambda(InnerNext_S)$. Because $\neg\Lambda(v_{\hat{S}}{}' = v_{\hat{S}})$, it also holds that $\neg\Lambda(AdvanceTime_S)$.
Therefore, it follows that $\neg\Lambda(Next_S)$ which implies that there exists a value $n \in \mathbb{N}$ so that $\neg\Lambda_{n+1}(Next_S)$.

By definition of χ, it holds that $\neg\Lambda_\chi(Next_S)$. While the difference between $s.x_\chi$ and $s.mappedClock^{x_\chi}$ might be greater than 1, there must be two values $d_1, d_2 \in \mathbb{N}_0$ with $d_1 + 1 = d_2$ so that a $Next_S$ is possible at d_2 but not at time d_1. Formally:

Define: $d_1, d_2 \in \mathbb{N}_0$ so that $d_1 + 1 = d_2$ and $d_1 \geq s.mappedClock^{x_\chi}$ and $d_2 \leq s.x_\chi$ and $T^{x_\chi}_{d_2}(\Lambda_\chi(s)) \xrightarrow{Next_S} T^{x_\chi}_{d_2}(\Lambda_\chi(t))$ but not $T^{x_\chi}_{d_1}(\Lambda_\chi(s)) \xrightarrow{Next_S} T^{x_\chi}_{d_1}(\Lambda_\chi(t))$

Such two points d_1, d_2 must exist because $s.mappedClock^{x_\chi} < s.x_\chi$ and at some point between these two values the step most become possible.

2.4.2.3. $T^{x_\chi}_{d_2}(\Lambda_\chi(s)) \approxeq s$

Proof: By definition of $T^{x_\chi}_{d_2}$ and $\Lambda_\chi(s)$.

2.4.2.4. $T^{x_\chi}_{d_2}(\Lambda_\chi(s)) \xrightarrow{[Next_S]^+_{v_S}} T^{x_\chi}_{d_2}(\Lambda_\chi(s))$

Proof: Because $[Next_S]^+_{v_S}$ allows stuttering steps.

2.4.2.5. $T^{x_\chi}_{d_2}(\Lambda_\chi(s)).x_\chi = d_2$

By definition of $T^{x_\chi}_{d_2}$

2.4.2.6. $d_2 \in ZTP^{x_\chi}(s)$

By definition of ZTP and the definition of d_1 and d_2 and 2.4.2.3 and 2.4.2.5.

2.4.2.7. Q.E.D.

Because $d_2 > s.mappedClock^{x_\chi}$ but $d_2 \in ZTP^{x_\chi}(s) \subseteq relZTP^{x_\chi}(s)$ it holds that $s.mappedClock^{x_\chi} \neq max(\{n \in relZTP^{x_\chi}(\ddot{s}) \mid n \leq s.x_\chi\})$ which contradicts $Inv_{S'}$. By 2.4.2.2 it follows that $\Lambda([Next_{\hat{S}}]_{v_{\hat{S}}})$.

2.4.3. Q.E.D.

Proof: Because the cases 2.4.1 and 2.4.2 cover all possibilities.

2.5. Q.E.D.

By 2.1 and because 2.2 and 2.3 and 2.4 and $Next_{S'} = AdvanceTime_{S'} \vee InnerNext_{S'}$ and $[Next_{S'}]_{v_{S'}} = Next_{S'} \vee (v'_{S'} = v_{S'})$.

3. Q.E.D.

By 1 and 2 it follows that $Spec_{S'} \Rightarrow \Lambda(Spec_{\hat{S}})$.

A.3.5. Proof of Lemma 4

We recall Lemma 4 which was defined on page 126:

Lemma 4. *In SpecificationI it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T^y_d(s))$$

Proof: $B^{LedgerTime}$ is defined by $TimeBounds$ of *PaymentChannelUser*. As the variable *LedgerTime* does not occur in the definition of $TimeBounds$, the definition of $TimeBounds$ is independent of *LedgerTime* and, thus, Assumption 4.1 holds for $B^{LedgerTime}$.
Similarly, B^{TxAge} is defined by $TxTimeBounds$ of *PaymentChannelUser*. As the variable $TxAge$ does not occur in the definition of $TxTimeBounds$, the definition of $TxTimeBounds$ is independent of $TxAge$ and, thus, Assumption 4.1 holds for B^{TxAge}.
In conclusion, Assumption 4.1 holds for all clocks of the specification.

A.3.6. Proof of Lemma 5

We recall Lemma 5 which was defined on page 126:

Lemma 5. *The definition of relaxed zone time points in* $relZTP^x$ *of SpecificationI*^{time} *meets Assumption 4.2, i.e. it holds that* $\forall x \in \mathcal{X}, s \in \hat{\Sigma} : ZTP^x(s) \subseteq relZTP^x(s)$.

Proof:
We first consider the *TxAge* clocks and then consider the clock *LedgerTime*.
The only steps of $NextI$ that depend on the value of a *TxAge* clock are steps in which a transaction is published that spends an output with a relative timelock. Such a step is possible only if the relative timelock has passed, i.e. if the respective *TxAge* clock for the transaction has reached the value of the relative timelock of the output that is being spent. Thus, the zone time points for *TxAge* are the values of relative timelocks. In *SpecificationI*, the only value used for relative timelocks is *TO_SELF_DELAY*. Because $relZTP^{TxAge}$ is defined as $\{TO\_SELF\_DELAY\}$, the lemma holds for the *TxAge* clocks.
For the proof for the clock *LedgerTime*, we use the observation that the set $ZTP^x(s)$ can be written as the union of sets $ZTP^x_A(s)$ for all subactions A of *NextI* using the following definition of ZTP^x_A.
This definition of ZTP^x_A is a modification of the definition of ZTP^x parameterized by a subaction A of *NextI*:

Definition 23 ($ZTP^x_A(s)$)**.**

$$
\begin{aligned}
ZTP^x_A : \Sigma &\to \mathcal{P}(\mathbb{N}) \\
s &\mapsto \{d \in \mathbb{N} \mid \exists s_0 \in \Sigma : s_0 \approxeq s \wedge \exists t \in \Sigma : \\
&\qquad\qquad s_0 \xrightarrow{[Next_S]^+_{v_S}} t \\
&\qquad\qquad \wedge\ t.x = d \\
&\qquad\qquad \wedge\ \exists u \in \Sigma : t \xrightarrow{A} u \wedge \neg(T^x_{d-1}(t) \xrightarrow{A} T^x_{u.x-1}(u)) \\
&\quad \}
\end{aligned}
$$

For some state s of *SpecificationI*, let $d \in ZTP^x_A(s)$ for a subaction A of *NextI*. The definition of $ZTP^x_A(s)$ implies that there is a step starting at a state t in which the clock x is set to value d and the step is not possible if clock x is set to the value $d-1$. The state t must be reachable from a state $s_0 \approxeq s$. We refer to the second state of the step that is possible at clock value d but not at clock value $d-1$ as state u. For the step $t \to u$, not to be possible,

there must be a condition that contains the clock x because the value of the clock x is the only difference between the states t and $T^x_{d-1}(t)$ This suggests the following approach to find all $d \in ZTP^x_A(s)$ for an action A: By inspecting the definition of action A, we find the conditions under which an A step exists that is enabled at clock value d but not at clock value $d-1$, and verify that in all states s from which a state that meets these conditions can be reached by steps of *NextI* it holds that $d \in relZTP^x(s)$.
We first consider the actions of the module *PaymentChannelUser*: There are six actions in the module that depend on the value of *LedgerTime* and three actions that depend on *TxAge*.

1. $ZTP^{LedgerTime}_{SendSignedCommitment}(s) \subseteq relZTP^{LedgerTime}(s)$
   The action *SendSignedCommitment* commits to new HTLCs by including them in the new commitment transaction that is sent. Outgoing HTLCs are included if they are in state NEW and if their timelock is greater than the current value of *LedgerTime*. For two consecutive points in time $d-1$ and d, a step of the action *SendSignedCommitment* is possible at time d but not at time $d-1$ if a *SendSignedCommitment* step is allowed at time $d-1$ that adds an HTLC but the same HTLC could not be added at time d because $d-1$ is smaller than the HTLC's timelock but d is equal to the HTLC's timelock. It follows that a long as there is an HTLC that is in state NEW, the set $relZTP^{LedgerTime}(s)$ must include the HTLC's timelock because at this point in time a new step might become possible that sends a commitment transaction without including this HTLC. This condition is fulfilled by the definition of *TimelockRegions* of the module *PaymentChannelUser*.
   Further, the set $relZTP^{LedgerTime}(s)$ must include the timelock of each HTLC that might come into state NEW. The only way for an HTLC to reach the state NEW is to be created by the action *AddAndSendOutgoingHTLC* in the module *HTLCUser*. This action creates an HTLC for an outgoing payment with the payment's timelock. Thus, the set $relZTP^{LedgerTime}(s)$ must include the timelock of each outgoing payment. This condition is fulfilled by the definition of *TimelockRegions* of the module *HTLCUser*.
   An outgoing payment is created if an incoming HTLC is received that is to be forwarded. The timelock of the newly created payment is taken from the *dataForNextHop* field in the message that was prepared by the original sender of the payment. Thus, the set $relZTP^{LedgerTime}(s)$ must include all the timelocks included in the *dataForNextHop* fields for payments that are already in process and, for multi-hop payments that are not yet in process, the set $relZTP^{LedgerTime}(s)$ must include the timelocks as they will appear in the *dataForNextHop* field. This condition is fulfilled by the definition of *TimelockRegions* of the module *HTLCUser*. It is possible to predict these timestamps because they are calculated only based on a payment's timelock and the payment's route which are known already in the initial states of the specification.
   Given that all these values are included in the set $relZTP^{LedgerTime}(s)$, the set $relZTP^{LedgerTime}(s)$ contains all points in time that are in the set $ZTP^{LedgerTime}_{SendSignedCommitment}(s)$ because the action *SendSignedCommitment* depends on *LedgerTime* only for choosing the HTLCs to add.
2. $ZTP^{LedgerTime}_{ReceiveSignedCommitment}(s) \subseteq relZTP^{LedgerTime}(s)$

Analogously to 1, *ReceiveSignedCommitment* accepts a commitment transaction that is being received only if the incoming HTLCs that are committed have a timelock that is greater than *LedgerTime*. An incoming HTLC can be committed only if it is in state NEW. As described above, the set $relZTP^{LedgerTime}(s)$ contains an HTLC's timelock already for all HTLCs that are or can come into state NEW. Therefore, the set $relZTP^{LedgerTime}(s)$ contains all points in time that are in the set $ZTP^{LedgerTime}_{ReceiveSignedCommitment}(s)$ because the action *ReceiveSignedCommitment* depends on *LedgerTime* only for deciding which HTLCs are allowed to be added in a commitment transaction being received.

3. $ZTP^{LedgerTime}_{ReceiveRevocationKey}(s) \subseteq relZTP^{LedgerTime}(s)$

   The action *ReceiveRevocationKey* depends on *LedgerTime* for defining which HTLCs are marked as COMMITTED. By definition of *ReceiveRevocationKey*, an incoming HTLC is only committed if it is in state SENT-COMMIT and has a timelock greater than *LedgerTime*. As described above, the set $relZTP^{LedgerTime}(s)$ contains an HTLC's timelock for all HTLCs that are or can come into state NEW. Additionally, the set $relZTP^{LedgerTime}(s)$ must also contain an HTLC's timelock if the HTLC is in state SENT-COMMIT or in one of the states RECV-COMMIT and PENDING-COMMIT which are preceding states of SENT-COMMIT. This condition is fulfilled by the definition of *TimelockRegions* of the module *PaymentChannelUser*. Therefore, the set $relZTP^{LedgerTime}(s)$ contains the timelocks of all HTLCs that are in state SENT-COMMIT or can come into this state.

4. $(ZTP^{LedgerTime}_{Cheat}(s) \cup ZTP^{LedgerTime}_{Punish}(s) \cup ZTP^{LedgerTime}_{ResolveHTLCAfterClose}(s)) \subseteq relZTP^{LedgerTime}(s)$

   The actions *Cheat*, *Punish*, and *ResolveHTLCAfterClose* all use an operator that finds transactions that can be used to spend outputs of published transactions and one of the transactions found is published. This operator depends on *LedgerTime* as well as $TxAge$ because outputs might be timelocked and only spendable after an absolute timelock or after a certain time after the transaction containing the output was published. If *LedgerTime* reaches the absolute timelock of an output or $TxAge$ reaches the relative timelock of an output, there is a new transaction in the set of publishable transactions and, thus, a step publishing this new transaction becomes possible. Consequently, the set $relZTP^{LedgerTime}(s)$ must contain absolute timelocks of outputs that might be spent by such a new transaction. This condition is fulfilled by the definition of *TimelockRegions* of the module *PaymentChannelUser* because *TimelockRegions* contains all absolute timelocks of outputs in the ledger.

   To account for the points in time at which an output of a transaction becomes spendable that has not been published yet, the set $relZTP^{LedgerTime}(s)$ must also include the absolute timelocks of transactions that are stored in the users' *Inventory* variables. This condition is fulfilled by the definition of *TimelockRegions* of the module *PaymentChannelUser*. Such absolute timelocks in transactions in the users' *Inventory* variables are created based on the absolute timelocks of uncommitted HTLCs. As shown in the steps above, the timelocks of these HTLCs are included by the definition of *TimelockRegions* of the module *PaymentChannelUser*.

5. $(ZTP^{TxAge}_{Cheat}(s) \cup ZTP^{TxAge}_{Punish}(s) \cup ZTP^{TxAge}_{ResolveHTLCAfterClose}(s)) \subseteq relZTP^{TxAge}(s)$

The actions *Cheat*, *Punish*, and *ResolveHTLCAfterClose* all use an operator that finds transactions that can be used to spend outputs of published transactions and one of the transactions found is published. This operator depends on *LedgerTime* as well as $TxAge$ because outputs might be timelocked and only spendable after an absolute timelock or after a certain time after the transaction containing the output was published. If $TxAge$ reaches the relative timelock of an output, there is a new transaction in the set of publishable transactions and, thus, a step publishing this new transaction becomes possible. Consequently, the set $relZTP^{TxAge}(s)$ must contain the relative timelocks of outputs that might be spent by such a new transaction. Because all relative timelocks in the specification equal the constant *TO_SELF_DELAY*, this condition is fulfilled by the definition of *AdvanceLedgerTime* which assumes that $relZTP^{TxAge}(s) = \{TO\_SELF\_DELAY\}$.

The module *HTLCUser* contains four actions that depend on *LedgerTime* and no actions that depend on *TxAge*:

6. $ZTP^{LedgerTime}_{AddAndSendOutgoingHTLC}(s) \subseteq relZTP^{LedgerTime}(s)$
   The action *AddAndSendOutgoingHTLC* adds a payment with the lowest timelock of all payments whose timelock is greater than the value of *LedgerTime*. Thus, a value d is in the set $ZTP^{LedgerTime}_{AddAndSendOutgoingHTLC}(s)$ if a payment cannot be added anymore at time d because the value of *LedgerTime* equals the payment's timelock and a payment with the next upcoming timelock becomes the payment with the lowest timelock of all payments whose timelock is greater than the value of *LedgerTime*. Because the definition of *TimelockRegions* of the module *HTLCUser* contains the timelock of each existing or future payment, all points in time that are in the set $ZTP^{LedgerTime}_{AddAndSendOutgoingHTLC}(s)$ are included in $relZTP^{LedgerTime}(s)$.
7. $ZTP^{LedgerTime}_{SendHTLCPreimage}(s) \subseteq relZTP^{LedgerTime}(s)$
   The action *SendHTLCPreimage* depends on *LedgerTime* and sends the preimage to an HTLC only if the HTLC's timelock plus the constant forward delta is greater than *LedgerTime*. The set $ZTP^{LedgerTime}_{SendHTLCPreimage}(s)$ is empty because, if a *SendHTLCPreimage* step exists one point in time, then any preceding point in time is also smaller than the HTLC's timelock plus the forward delta.
8. $ZTP^{LedgerTime}_{SendHTLCFail}(s) \subseteq relZTP^{LedgerTime}(s)$
   The action *SendHTLCFail* depends on *LedgerTime* to fail an HTLC that has been committed if the HTLC has timed out. Thus, the set $ZTP^{LedgerTime}_{SendHTLCFail}(s)$ contains the timelock of an HTLC that is in state COMMITTED. As the set $relZTP^{LedgerTime}(s)$ contains the timelock of all HTLCs that are committed and can come into state COMMITTED, if follows that $ZTP^{LedgerTime}_{SendHTLCFail}(s) \subseteq relZTP^{LedgerTime}(s)$.
9. $ZTP^{LedgerTime}_{ReceiveHTLCFail}(s) \subseteq relZTP^{LedgerTime}(s)$
   The action *ReceiveHTLCFail* marks an HTLC only as failed if the HTLC's timelock is smaller than or equal to *LedgerTime*. This is the same condition as for the action *SendHTLCFail* and, by the same argument as in 8, if holds that $ZTP^{LedgerTime}_{ReceiveHTLCFail}(s) \subseteq relZTP^{LedgerTime}(s)$.

The module *ChannelEnvironment* contains no actions that depends on *LedgerTime* or $TxAge$.
Finally, the module *LedgerTime* also depends on *LedgerTime*.

10. $ZTP^{LedgerTime}_{AdvanceLedgerTime}(s) \subseteq relZTP^{LedgerTime}(s)$

    A step of *AdvanceLedgerTime* that is possible when *LedgerTime* equals a value d is not possible at time $d - 1$ if there is a time bound at time $d - 1$ that prevents time from advancing. Besides the time bounds, a step of *AdvanceLedgerTime* is always possible. Thus, the set $ZTP^{LedgerTime}_{AdvanceLedgerTime}(s)$ equals the set $\{b + 1 \mid b \in B^x(s)\}$. As the definition of *relZTP* in *SpecificationI*^{time} explicitly includes the set $\{b + 1 \mid b \in B^x(s)\}$, it holds that $ZTP^{LedgerTime}_{AdvanceLedgerTime}(s) \subseteq relZTP^{LedgerTime}(s)$.

11. Q.E.D.

    Because the union of all sets $ZTP^{LedgerTime}_{A}(s)$ for subactions A of *NextI* equals the set $ZTP^{LedgerTime}(s)$, it follows that $ZTP^{LedgerTime}(s) \subseteq relZTP^{LedgerTime}(s)$.

A.3.7. Proof of Lemma 6

We recall Lemma 6 which was defined on page 128:

Lemma 6. *For any two states s, t of SpecificationI and all clocks $x \in \mathcal{X}$ (i.e., LedgerTime and TxAge), it holds that*

$$NextI \Rightarrow relZTP^{x\prime} \subseteq relZTP^{x}$$

To prove this lemma, we first consider the simple case of the *TxAge* clocks and then consider the the clock *LedgerTime*.

The set $relZTP^{TxAge}$ is defined as $\{TO\_SELF\_DELAY\}$ independently of the current state. Therefore, the lemma trivially holds for the *TxAge* clocks.

The specification defines the set $relZTP^{LedgerTime}$ as the union of the sets *TimelockRegions* of the modules *PaymentChannelUser* and the module *HTLCUser* and the set $\{t + 1 : t \in TimeBounds\}$. These sets are again defined as the union of smaller sets.

The set $PCU(c, u)!TimelockRegions$ is a union of four sets:

- A set T_1 that contains the HTLC's timelock for all HTLCs that are in states $H_1 = \{$NEW, RECV-COMMIT, PENDING-COMMIT, SENT-COMMIT, COMMITTED$\}$.
- A set T_2 that contains for all HTLCs that are in states $H_2 = \{$NEW, RECV-COMMIT, PENDING-COMMIT, SENT-COMMIT, COMMITTED, FULFILLED, OFF-TIMEDOUT, SENT-REMOVE, PENDING-REMOVE, RECV-REMOVE$\}$ the HTLC's timelock + *FORWARD_DELTA* + 1.
- A set T_3 that contains the absolute timelock of all transactions in the user u's inventory.
- A set T_4 that contains the absolute timelock of all transactions in the ledger.

We refer to these sets as $PCU(c,u)!T_1$ to $PCU(c,u)!T_4$.

The set $H(c,u)!TimelockRegions$ is a union of three sets:

- A set T_1 that contains for each payment in *NewPayments* the timelock and the timelock +*FORWARD_DELTA* + 1 for each HTLC on the payment's route.
- A set T_2 that contains the timelock of each HTLC and the timelock + *FORWARD_DELTA* + 1 for which an `update_add_htlc` message is contained in *ChannelMessages*. This set includes also the HTLCs that are contained in the message as nested payload to be forwarded.
- A set T_3 that contains the timelock and the timelock +*FORWARD_DELTA* + 1 for each HTLC that has not been fulfilled.

We refer to these sets as $H(c,u)!T_1$ to $H(c,u)!T_3$.

The set $PCU(c,u)!TimeBounds$ is the union of three sets:

- A set B_1 that contains an HTLC's timelock -1 for all incoming HTLCs that are fulfilled but not in a final state.
- A set B_2 that contains an HTLC's timelock plus the forward delta -1 for all outgoing HTLCs that are not fulfilled.
- A set B_3 that contains the value MAX_TIME and, if the protocol state does not equal `closed`, the value $MAX\_TIME - TO\_SELF\_DELAY$.

We refer to these sets as $PCU(c,u)!B_1$ to $PCU(c,u)!B_3$.

In the proof, we show that for each of these sets it holds that the set in state t is a subset of the set in state s.

Proof:

1. $AdvanceTimeI \Rightarrow relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$

   Proof: Because the definition of $relZTP^{LedgerTime}$ does not depend on the value of a clock variable but uses only non-clock variables that are unchanged by $AdvanceTimeI$.

We look at steps that do not create a new HTLC.

2. Case: $NextI \wedge \neg H(c,u)!AddAndSendOutgoingHTLC \wedge \neg H(c,u)!ReceiveUpdateAddHTLC$

   2.1. $\forall c \in C, u \in U : PCU(c,u)!TimelockRegions' \subseteq PCU(c,u)!TimelockRegions$

   2.1.1. $\forall c \in C, u \in U : PCU(c,u)!T_1' \subseteq PCU(c,u)!T_1$

   Proof: As shown in Section 3.3.2.7, an existing HTLC can only reach one of the states that are in H_1 from a state that is already in H_1.

   2.1.2. $\forall c \in C, u \in U : PCU(c,u)!T_2' \subseteq PCU(c,u)!T_2$

   Proof: As shown in Section 3.3.2.7, an existing HTLC can only reach one of the states that are in H_2 from a state that is already in H_2.

   2.1.3. $\forall c \in C, u \in U : PCU(c,u)!T_3' \subseteq PCU(c,u)!T_3 \cup PCU(c,u)!T_1$

Proof: Because the transactions with absolute timelocks are only added to a user's inventory if the HTLC reaches the states SENT-COMMIT or RECV-COMMIT. The absolute timelocks of these HTLCs are contained in $PCU(c,u)!T_1$.

2.1.4. $\forall c \in C, u_1 \in U : PCU(c,u_1)!T_4' \subseteq PCU(c,u_1)!T_4 \cup \bigcup_{u_2 \in U}(PCU(c,u_2)!T_3)$

Proof: Because each transaction that is added to the ledger has previously been in one of the users' inventories.

2.1.5. Q.E.D.

Proof: By definition of $PCU(c,u)!TimelockRegions$

2.2. $\forall c \in C, u \in U : H(c,u)!TimelockRegions' \subseteq H(c,u)!TimelockRegions$

2.2.1. $\forall c \in C, u \in U : H(c,u)!T_1' \subseteq H(c,u)!T_1$

Proof: Because the only action that adds a payment to *NewPayments* is *H(c,u)!ReceiveUpdateAddHTLC* and, by 2, it holds that $\neg H(c,u)!ReceiveUpdateAddHTLC$.

2.2.2. $\forall c \in C, u \in U : H(c,u)!T_2' \subseteq H(c,u)!T_2$

Proof: Because the only action that creates an `update_add_htlc` message is *H(c,u)!AddAndSendOutgoingHTLC* and, by 2, it holds that $\neg H(c,u)!AddAndSendOutgoingHTLC$.

2.2.3. $\forall c \in C, u \in U : H(c,u)!T_3' \subseteq H(c,u)!T_3$

Proof: Because an HTLC that has not been fulfilled in state t has already not been fulfilled in state s.

2.2.4. Q.E.D.

Proof: By definition of $H(c,u)!TimelockRegions$

2.3. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!TimeBounds\}' \subseteq relZTP^{LedgerTime}$

Proof:

2.3.1. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_1\}' \subseteq relZTP^{LedgerTime}$

Proof: If an HTLC's timelock is in the set $\{t+1 \mid t \in PCU(c,u)!B_1\}'$ then it has either already been fulfilled before the step or it is fulfilled in the current step. In the first case, the HTLC's timelock is in the set $relZTP^{LedgerTime}$ because it is in the set $\{t+1 \mid t \in PCU(c,u)!B_1\}$. In the second case, the HTLC's timelock is in the set $relZTP^{LedgerTime}$ because it is in the set $PCU(c,u)!T_1$.

2.3.2. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_2\}' \subseteq relZTP^{LedgerTime}$

Proof: If a value is in the set $\{t+1 \mid t \in PCU(c,u)!B_2\}'$ then it is also in the set $\{t+1 \mid t \in PCU(c,u)!B_2\}$ unless the HTLC's state before the step was NEW. If the HTLC's state before the step was NEW, the value is in the set $PCU(c,u)!T_1$. In either case, it follows that the value is in the set $relZTP^{LedgerTime}$.

2.3.3. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_3\}' \subseteq relZTP^{LedgerTime}$

Proof: There are only two different values that the set $PCU(c,u)!B_3$ can assume: Before user u is in the protocol state `closed`, the set has the value $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ and when user u is in the protocol state `closed`, the set has the value $\{MAX\_TIME\}$. Because the protocol state `closed` is the final protocol state that is not changed anymore, the value of the set

$PCU(c,u)!B_3$ changes only from $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ to $\{MAX\_TIME\}$.

2.3.4. Q.E.D.

Proof: By definition of $PCU(c,u)!TimeBounds$.

2.4. Q.E.D.

Proof: By definition of $relZTP^{LedgerTime}$

3. Case: $H(c,u)!AddAndSendOutgoingHTLC \vee H(c,u)!ReceiveUpdateAddHTLC$

Proof: These actions add a an outgoing resp. incoming HTLC with state NEW. Further, $H(c,u)!AddAndSendOutgoingHTLC$ sends an `update_add_htlc` message and $H(c,u)!ReceiveUpdateAddHTLC$ adds a new payment to *NewPayments*.

3.1. $\forall c \in C, u \in U : PCU(c,u)!TimelockRegions' \subseteq relZTP^{LedgerTime}$

3.1.1. $\forall c \in C, u \in U : PCU(c,u)!T_1' \subseteq relZTP^{LedgerTime}$

Proof: Because the state of an HTLC does not change in the step, we only have to consider that $PCU(c,u)!T_1'$ contains the timelock of an HTLC in state NEW. If $H(c,u)!AddAndSendOutgoingHTLC$ then this timelock is the timelock of a payment in $NewPayments$ and, by definition of $H(c,u)!T_1$, the HTLC's timelock is in the set $relZTP^{LedgerTime}$. If $H(c,u)!ReceiveUpdateAddHTLC$ this timelock is the timelock of an HTLC in an `update_add_htlc` message and, by definition of $H(c,u)!T_2$, the HTLC's timelock is in the set $relZTP^{LedgerTime}$.

3.1.2. $\forall c \in C, u \in U : PCU(c,u)!T_2' \subseteq relZTP^{LedgerTime}$

Proof: Analogously to 3.1.1 because the sets $H(c,u)!T_1$ and $H(c,u)!T_2$ also contain an HTLC's timelock $+FORWARD\_DELTA + 1$.

3.1.3. $\forall c \in C, u \in U : PCU(c,u)!T_3' = PCU(c,u)!T_3$

Proof: Because the set of transactions in a user's inventory is unchanged in the step.

3.1.4. $\forall c \in C, u_1 \in U : PCU(c,u_1)!T_4' = PCU(c,u_1)!T_4$

Proof: Because the set of transactions in the ledger is unchanged in the step.

3.1.5. Q.E.D.

Proof: By definition of $PCU(c,u)!TimelockRegions$

3.2. $\forall c \in C, u \in U : H(c,u)!TimelockRegions' \subseteq H(c,u)!TimelockRegions$

3.2.1. $\forall c \in C, u \in U : H(c,u)!T_1' \subseteq H(c,u)!T_1 \cup H(c,u)!T_2$

Proof: The only action that adds a payment to *NewPayments* is *H(c,u)!ReceiveUpdateAddHTLC*. Thus, if $H(c,u)!AddAndSendOutgoingHTLC$ it holds that $H(c,u)!T_1' = H(c,u)!T_1$. If $H(c,u)!ReceiveUpdateAddHTLC$, it holds that $H(c,u)!T_1' \subseteq H(c,u)!T_2$ because the absolute timelock of the payment is copied from the `update_add_htlc` message that is received.

3.2.2. $\forall c \in C, u \in U : H(c,u)!T_2' \subseteq H(c,u)!T_2 \cup H(c,u)!T_1$

Proof: The only action that creates an `update_add_htlc` message is *H(c,u)!AddAndSendOutgoingHTLC*. Thus, if $H(c,u)!ReceiveUpdateAddHTLC$, it holds that $H(c,u)!T_2' = H(c,u)!T_2$. If $H(c,u)!AddAndSendOutgoingHTLC$, it holds that $H(c,u)!T_2' \subseteq H(c,u)!T_1$ because, by definition of $H(c,u)!T_1$, all

values that are calculated for a `update_add_htlc` message are contained in the set $H(c,u)!T_1$.

3.2.3. $\forall c \in C, u \in U : H(c,u)!T_3' \subseteq H(c,u)!TimelockRegions$

Proof: Because the state of an HTLC does not change in the step, we only have to consider that $H(c,u)!T_3'$ contains the timelock of an HTLC in state NEW. If $H(c,u)!AddAndSendOutgoingHTLC$ then this timelock is the timelock of a payment in $NewPayments$ and, by definition of $H(c,u)!T_1$, the HTLC's timelock is in the set $H(c,u)!TimelockRegions$. If $H(c,u)!ReceiveUpdateAddHTLC$ this timelock is the timelock of an HTLC in an `update_add_htlc` message and, by definition of $H(c,u)!T_2$, the HTLC's timelock is in the set $H(c,u)!TimelockRegions$.

3.2.4. Q.E.D.

Proof: By definition of $H(c,u)!TimelockRegions$

3.3. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!TimeBounds\}' \subseteq relZTP^{LedgerTime}$

Proof:

3.3.1. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_1\}' = \{t+1 \mid t \in PCU(c,u)!B_1\}$

Proof: By 3, an HTLC is not fulfilled in this step. Thus, an HTLC that is fulfilled after the step must already have been fulfilled at the beginning of the step.

3.3.2. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_2\}' \subseteq relZTP^{LedgerTime}$

Proof: Because the state of an HTLC does not change in the step, we only have to consider that $\{t+1 \mid t \in PCU(c,u)!B_2\}'$ contains the timelock of an HTLC in state NEW. If $H(c,u)!AddAndSendOutgoingHTLC$ then this timelock is the timelock of a payment in $NewPayments$ and, by definition of $H(c,u)!T_1$, the HTLC's timelock is in the set $relZTP^{LedgerTime}$. If $H(c,u)!ReceiveUpdateAddHTLC$ this timelock is the timelock of an HTLC in an `update_add_htlc` message and, by definition of $H(c,u)!T_2$, the HTLC's timelock is in the set $relZTP^{LedgerTime}$.

3.3.3. $\forall c \in C, u \in U : \{t+1 \mid t \in PCU(c,u)!B_3\}' \subseteq relZTP^{LedgerTime}$

Proof: There are only two different values that the set $PCU(c,u)!B_3$ can assume: Before user u is in the protocol state `closed`, the set has the value $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ and when user u is in the protocol state `closed`, the set has the value $\{MAX\_TIME\}$. Because the protocol state `closed` is the final protocol state that is not changed anymore, the value of the set $PCU(c,u)!B_3$ changes only from $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ to $\{MAX\_TIME\}$.

3.3.4. Q.E.D.

Proof: By definition of $PCU(c,u)!TimeBounds$.

3.4. Q.E.D.

Proof: By definition of $relZTP^{LedgerTime}$

4. Q.E.D.

By 2 and 3.

A.3.8. Example Trace of *SpecificationI*$^{time,\ single}$ Mapped to *SpecificationII*^{single} by the Mapping f

Table A.14 shows a behavior in which a channel is opened, an off-chain payment is made, and the channel is closed.

A.3.9. Definition of Mapping m_c to Map a State of *SpecificationI*^{time} to a State of *SpecificationI*$^{time,\ single}$

Figure A.1 defines the mapping m_c. The context of this code snippet is the specification *SpecificationIItoIIa* which is a specification that extends *SpecificationI*^{time} and creates an instance of *SpecificationI*$^{time,\ single}$. The mapping m_c is used to define the values of the variables of the instance of *SpecificationI*$^{time,\ single}$. The code is defined in the TLA$^+$ specification in the file *SpecificationIItoIIa.tla*.

A.3.10. Proof of Lemma 7

We recall Lemma 7 which was defined on page 141:

Lemma 7. *Given a state s of SpecificationI^{time}, it holds that for every step $s \rightarrow t$ of an action of PaymentChannelUser or HTLCUser for channel c, in all channels $c' \neq c$ the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an instance of ChannelEnvironment for channel c' or a or a step of HTLCUser for channel c' and the same user as in channel c.*

Proof: Let $s \rightarrow t$ be a step in *SpecificationI*^{time} of an action of the modules *PaymentChannelUser* or *HTLCUser* for channel c.
The variables that are specific to channel c' and the variables of users u who are not part of channel c are unaffected by a step in channel c and, thus, these variables are unchanged in the step $m_{c'}(s) \rightarrow m_{c'}(t)$.
Variables that are left to discuss because they are possibly changed in the step $m_{c'}(s) \rightarrow m_{c'}(t)$ are the global variables v_{g} and the user specific variables v_u for users u who are both in channel c' and channel c. These variables are: *Messages*, *LedgerTx*, *LedgerTime*, *TxAge*, *UserPreimageInventory[u]*, *UserPaymentSecretForPreimage[u]*, *UserChannelBalances[u]*, *UserPayments[u]*, *UserNewPayments[u]*, *UserHonest[u]*, *UserExtBalance[u]*, and *UserRequestedInvoices[u]*.
The variables *LedgerTx* and *TxAge* are reduced by $m_{c'}$ to values that are specific to channel c'. Because transactions are specific to at most one channel, a step in channel c does not affect the value of *LedgerTx* and *TxAge* in the step $m_{c'}(s) \rightarrow m_{c'}(t)$ and these variables are unchanged.
The variable *LedgerTime* cannot be changed in step $s \rightarrow t$ because $s \rightarrow t$ is a step of an action of the modules *PaymentChannelUser* or *HTLCUser*. Because, by Lemma 21, the set $relZTP^{LedgerTime}$ in *SpecificationI*$^{time,\ single}$ can only become smaller during the step

	Step of $SpecificationI^{time,\ single}$	Step of $SpecificationII^{single}$
1	PCU(1,1)!SendOpenChannel	
2	PCU(1,2)!SendAcceptChannel	
3	PCU(1,1)!CreateFundingTransaction	
4	PCU(1,1)!SendSignedFirstCommitTransaction	
5	PCU(1,2)!ReplyWithFirstCommitTransaction	
6	PCU(1,1)!ReceiveCommitTransaction	
7	PCU(1,1)!PublishFundingTransaction	AC(1)!OpenPaymentChannel
8	PCU(1,2)!NoteThatFundingTransactionPublished	
9	PCU(1,1)!SendNewRevocationKey	
10	PCU(1,2)!SendNewRevocationKey	
11	PCU(1,1)!ReceiveNewRevocationKey	
12	PCU(1,2)!ReceiveNewRevocationKey	
13	HU(1,1)!RequestInvoice	HU(1,1)!RequestInvoice
14	HU(1,2)!GenerateAndSendPaymentHash	HU(1,2)!GenerateAndSendPaymentHash
15	HU(1,1)!ReceivePaymentHash	HU(1,1)!ReceivePaymentHash
16	HU(1,1)!AddAndSendOutgoingHTLC	HU(1,1)!AddAndSendOutgoingHTLC
17	HU(1,2)!ReceiveUpdateAddHTLC	HU(1,2)!ReceiveUpdateAddHTLC
18	PCU(1,1)!SendSignedCommitment	
19	PCU(1,2)!ReceiveSignedCommitment	
20	PCU(1,2)!RevokeAndAck	
21	PCU(1,1)!ReceiveRevocationKey	
22	PCU(1,2)!SendSignedCommitment	
23	PCU(1,1)!ReceiveSignedCommitment	
24	PCU(1,1)!RevokeAndAck	
25	PCU(1,2)!ReceiveRevocationKey	AC(1)!UpdatePaymentChannel
26	HU(1,2)!SendHTLCPreimage	HU(1,2)!SendHTLCPreimage
27	HU(1,1)!ReceiveHTLCPreimage	HU(1,1)!ReceiveHTLCPreimage
28	PCU(1,2)!SendSignedCommitment	
29	PCU(1,1)!ReceiveSignedCommitment	
30	PCU(1,1)!RevokeAndAck	
31	PCU(1,2)!ReceiveRevocationKey	
32	PCU(1,1)!SendSignedCommitment	
33	PCU(1,2)!ReceiveSignedCommitment	
34	PCU(1,2)!RevokeAndAck	AC(1)!UpdatePaymentChannel
35	PCU(1,1)!ReceiveRevocationKey	
36	PCU(1,2)!CloseChannel	AC(1)!SetOnChainHTLCsAndCheater
37	PCU(1,1)!NoteThatOtherUserClosedHonestly	AC(1)!CommitHTLCsOnChain
38	PCU(1,1)!Terminate	
39	PCU(1,2)!Terminate	AC(1)!ClosePaymentChannel

Table A.14.: Trace of an execution in which a channel is opened, an off-chain payment is made, and the channel is closed.

$$
\begin{array}{l}
SpecificationIIa(CID) \triangleq \text{INSTANCE } SpecificationIIa \text{ WITH} \\
\quad ActiveChannels \leftarrow \{CID\}, \\
\quad Messages \leftarrow \{msg \in Messages : \\
\qquad \land \ \lor msg.sender \in \{NameForUserID[uid] : uid \in UsersOfChannelSet(CID)\} \\
\qquad\quad \lor msg.recipient \in \{NameForUserID[uid] : uid \in UsersOfChannelSet(CID)\} \\
\qquad \land \ \lor \exists\, pmt \in AllInitialPaymentsOverChannel(CID) : \\
\qquad\qquad\qquad msg.sender \in \{pmt.path[1],\ pmt.path[Len(pmt.path)]\} \\
\qquad\quad \lor \exists\, pmt \in AllInitialPaymentsOverChannel(CID) : \\
\qquad\qquad\qquad msg.recipient \in \{pmt.path[1],\ pmt.path[Len(pmt.path)]\}\}, \\
\quad LedgerTx \leftarrow TranslateTimeOfTxs(CID,\ ChannelLedgerTx(CID)), \\
\quad LedgerTime \leftarrow ChannelMappedLedgerTime[CID], \\
\quad TxAge \leftarrow [id \in \{tx.id : tx \in ChannelLedgerTx(CID)\} \quad \mapsto TxAge[id]], \\
\quad UserPreimageInventory \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad UserPreimageInventory[u] \cap RelevantPreimages(CID,\ u)], \\
\quad UserPaymentSecretForPreimage \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad [preimage \in \text{DOMAIN } UserPaymentSecretForPreimage[u] \cap \\
\qquad RelevantPreimages(CID,\ u) \mapsto UserPaymentSecretForPreimage[u][preimage]]], \\
\quad UserChannelBalances \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad [c \in \{CID\} \mapsto UserChannelBalances[u][CID]]], \\
\quad UserInitialPayments \leftarrow [u \in UserIDs \mapsto InitialPaymentsOverChannel(u,\ CID)], \\
\quad UserPayments \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad UserPaymentsOverChannel(u,\ CID)], \\
\quad UserNewPayments \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad UserNewPaymentsOverChannel(u,\ CID)], \\
\quad UserHonest \leftarrow [u \in UsersOfChannelSet(CID) \mapsto UserHonest[u]], \\
\quad UserInitialBalance \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad \text{IF } CID \in \text{DOMAIN } UserChannelFundingAmount[u] \\
\qquad\qquad\qquad \text{THEN } UserChannelFundingAmount[u][CID] \text{ ELSE } 0], \\
\quad UserExtBalance \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad\qquad ChannelSpecificExtBalance(CID,\ u)], \\
\quad UserRequestedInvoices \leftarrow [u \in UsersOfChannelSet(CID) \mapsto \\
\qquad MockedRequestedInvoices(\text{UNION} \\
\qquad\qquad \{InitialPaymentsOverChannel(o,\ CID) : o \in \{o \in UserIDs : o \neq u\}\}, \\
\qquad \text{UNION } \{UserNewPayments[o] : o \in \{o \in UserIDs : o \notin UsersOfChannelSet(CID)\}\}, \\
\qquad NameForUserID[u], \\
\qquad NameForUserID[\text{CHOOSE } o \in UsersOfChannelSet(CID) : o \neq u],\ CID)], \\
\quad ChannelMessages \leftarrow [c \in \{CID\} \mapsto ChannelMessages[c]], \\
\quad ChannelUsedTransactionIds \leftarrow [c \in \{CID\} \mapsto ChannelUsedTransactionIds[c]], \\
\quad ChannelPendingBalance \leftarrow [c \in \{CID\} \mapsto ChannelPendingBalance[c]], \\
\quad ChannelUserBalance \leftarrow [c \in \{CID\} \mapsto ChannelUserBalance[c]], \\
\quad ChannelUserState \leftarrow [c \in \{CID\} \mapsto ChannelUserState[c]], \\
\quad ChannelUserVars \leftarrow [c \in \{CID\} \mapsto ChannelUserVars[c]], \\
\quad ChannelUserDetailVars \leftarrow [c \in \{CID\} \mapsto ChannelUserDetailVars[c]], \\
\quad ChannelUserInventory \leftarrow [c \in \{CID\} \mapsto ChannelUserInventory[c]]
\end{array}
$$

Figure A.1.: Formal definition in TLA$^+$ of the mapping m_c (CID is used instead of c) that creates a state of $SpecificationI^{time,\ single}$ based on a state of $SpecificationI^{time}$. The operators used are defined in Figs. A.2 to A.8

$$
\begin{aligned}
&MockedRequestedInvoices(OtherUsersInitialPayments,\ OtherUsersNewPayments,\\
&\qquad\qquad User,\ OtherUser,\ c) \triangleq\\
&\quad \{payment : payment \in\\
&\qquad \{payment \in OtherUsersInitialPayments :\\
&\qquad\quad \exists\, pmt \in OtherUsersNewPayments\ :\\
&\qquad\quad \land\ payment.id = pmt.id\\
&\qquad\quad \land\ payment.amount = pmt.amount\\
&\qquad\quad \land\ payment.path[Len(payment.path)] = User\\
&\qquad\quad \land\ pmt.invoiceRequested\}\\
&\quad \} \cup\\
&\quad \{payment \in AllInitialPaymentsOverChannel(c) :\\
&\qquad \land\ payment.path[1] \neq OtherUser\\
&\qquad \land\ payment.path[Len(payment.path)] = User\\
&\qquad \land\ \neg\exists\, uid \in \text{DOMAIN}\ UserNewPayments :\\
&\qquad\quad \land\ payment.path[1] = NameForUserID[uid]\\
&\qquad\quad \land\ \exists\, pmt \in UserNewPayments[uid] : pmt.id = payment.id\\
&\quad \}
\end{aligned}
$$

Figure A.2.: Definition of the operator *MockedRequestedInvoices* used in Fig. A.1

$$
\begin{aligned}
&RelevantPreimages(c,\ u) \triangleq\\
&\quad \{PreimageForPayment(pmt) : pmt \in \{pmt \in AllInitialPaymentsOverChannel(c) :\\
&\qquad pmt.path[Len(pmt.path)] = NameForUserID[u]\}\}\\
&\quad \cup \{htlc.hash : htlc \in\\
&\qquad ChannelUserVars[c][u].IncomingHTLCs \cup\\
&\qquad ChannelUserVars[c][u].OutgoingHTLCs\}
\end{aligned}
$$

Figure A.3.: Definition of the operator *RelevantPreimages* used in Fig. A.1

$m_{c'}(s) \rightarrow m_{c'}(t)$, the value of *LedgerTime* in state $m_{c'}(t)$ must equal $m_{c'}(s).LedgerTime$, i.e. the value of *LedgerTime* does not change in the step $m_{c'}(s) \rightarrow m_{c'}(t)$.
The remaining variables are changed by the following actions of the modules *PaymentChannelUser* and *HTLCUser*: *PublishFundingTransaction*, *ResolveHTLCAfterClose*, *NoteThatHTLCFulfilledOnChainByOtherUser*, *RequestInvoice*, *GenerateAndSendPaymentHash*, *Receive-*

$$
\begin{aligned}
&ChannelSpecificExtBalance(c,\ u) \triangleq\\
&\quad \text{IF}\ UserIsFunderOfChannel(c,\ u) \land ChannelUserState[c][u] \in\\
&\qquad \{\text{“init”, “open-sent-open-channel”, “open-funding-created”,}\\
&\qquad\ \text{“open-sent-commit-funder”, “open-recv-commit”}\}\\
&\quad \text{THEN}\ UserChannelFundingAmount[u][c]\\
&\quad \text{ELSE IF}\ LedgerTime < TERMINATION \lor \neg UserHonest[u]\\
&\quad \text{THEN}\ 0\\
&\quad \text{ELSE}\ UserOnChainBalance(ChannelLedgerTx(c),\ TERMINATION,\\
&\qquad\qquad TO\_SELF\_DELAY,\ u)
\end{aligned}
$$

Figure A.4.: Definition of the operator *ChannelSpecificExtBalance* used in Fig. A.1

```
UserPaymentsOverChannel(u, c) ≜
  {pmt ∈ UserPayments[u] :
    ∃ iPmt ∈ UserInitialPayments[pmt.sender]
                ∪ UserInitialPayments[pmt.receiver] :
      ∧ iPmt.id = pmt.id
      ∧  ∨ iPmt.path[2] = NameForUserID[
               CHOOSE o ∈ UsersOfChannelSet(c) : o ≠ u]
         ∨ iPmt.path[Len(iPmt.path) − 1] = NameForUserID[
               CHOOSE o ∈ UsersOfChannelSet(c) : o ≠ u]
  }
```

Figure A.5.: Definition of the operator *UserPaymentsOverChannel* used in Fig. A.1

```
UserNewPaymentsOverChannel(u, c) ≜
  {pmt ∈ UserNewPayments[u] :
      ∨ "path" ∉ DOMAIN pmt
      ∨ pmt.path[2] = NameForUserID[
            CHOOSE o ∈ UsersOfChannelSet(c) : o ≠ u]
  }
```

Figure A.6.: Definition of the operator *UserNewPaymentsOverChannel* used in Fig. A.1

```
PredecessorsOfTx(tx) ≜
  CHOOSE predecessors ∈ SUBSET LedgerTx :
    ∧  ∀ pred ∈ predecessors :
         ∀ input ∈ pred.inputs :
           ∃ prePred ∈ predecessors :
             input.parentId = prePred.id
    ∧  ∀ input ∈ tx.inputs :
         ∃ pred ∈ predecessors :
           input.parentId = pred.id

ChannelLedgerTx(ChannelID) ≜
  {tx ∈ LedgerTx :
    ∨ tx.id = ChannelID
    ∨ ChannelID ∈ {pred.id : pred ∈ PredecessorsOfTx(tx)}
  }
```

Figure A.7.: Definition of the operator *ChannelLedgerTx* used in Fig. A.1

```
TranslateTimeOfTxs(ChannelID, Txs) ≜
  {[tx EXCEPT !.absTimelock =
    IF @ ∈ DOMAIN ChannelMappedLedgerTimeHistory[ChannelID]
    THEN ChannelMappedLedgerTimeHistory[ChannelID][@]
    ELSE @] : tx ∈ Txs}
```

Figure A.8.: Definition of the operator *TranslateTimeOfTxs* used in Fig. A.1

PaymentHash, *IgnoreInvoiceRequest*, *SendHTLCPreimage*, *ReceiveHTLCPreimage*, *AddAndSendOutgoingHTLC*, and *ReceiveUpdateAddHTLC*. In the following, we prove for each of these actions that a step $s \to t$ of these actions in channel c is a step $m_{c'}(s) \to m_{c'}(t)$ in channel c' and either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for the same user.

1. If $s \to t$ is a step of *PublishFundingTransaction* of user u in channel c, then $m_{c'}(s) \to m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The variables that are changed by *PublishFundingTransaction* and are not specific to channel c are *UserExtBalance[u]* and *UserChannelBalances[u]*. By definition of the mapping m_c, both variables are unchanged in step $m_{c'}(s) \to m_{c'}(t)$ because they are mapped to channel specific values. As all other variables remain unchanged, $m_{c'}(s) \to m_{c'}(t)$ is a step of *UserOpensPaymentChannel* if user u is part of channel c' and a stuttering step otherwise.

2. If $s \to t$ is a step of *ResolveHTLCAfterClose* or *SendHTLCPreimage* of user u in channel c, then $m_{c'}(s) \to m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The variables that are changed by *ResolveHTLCAfterClose* and *SendHTLCPreimage* and are not specific to channel c are *UserPayments[u]* and *UserChannelBalances[u]*. By definition of m_c, both variables are unchanged in step $m_{c'}(s) \to m_{c'}(t)$ because they are mapped to channel specific values.

3. If $s \to t$ is a step of *NoteThatHTLCFulfilledOnChainByOtherUser* or *ReceiveHTLCPreimage* of user u in channel c, then $m_{c'}(s) \to m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The variables that are changed by *NoteThatHTLCFulfilledOnChainByOtherUser* and *ReceiveHTLCPreimage* and are not specific to channel c are *UserPreimageInventory[u]*, *UserPayments[u]* and *UserChannelBalances[u]*. Because the variables *UserPayments[u]* and *UserChannelBalances[u]* are mapped by m_c to channel-specific values, these variables are unchanged in the step $m_{c'}(s) \to m_{c'}(t)$. By definition of m_c, these variables are unchanged in step $m_{c'}(s) \to m_{c'}(t)$ if user u is not part of channel c'. Further, by definition of m_c, the variable *UserPreimageInventory[u]* is unchanged if the preimage that is being read from the blockchain is not in the set $R_{u,c'}$.
   We distinguish the cases that user u is the receiver of the payment associated with the fulfilled HTLC and that user u is not the receiver: If user u is the receiver of the payment associated with the fulfilled HTLC, then the HTLC is not part of another channel of user u and, thus, the preimage is not in the set $R_{u,c'}$ and also the variable *UserPreimageInventory[u]* is unchanged. Thus, the step $m_{c'}(s) \to m_{c'}(t)$ is a stuttering step.
   If user u is not the receiver of the payment associated with the fulfilled HTLC, the variable *UserPreimageInventory[u]* changes only in step $m_{c'}(s) \to m_{c'}(t)$ if the preimage of the fulfilled HTLC is in $R_{u,c'}$. If *UserPreimageInventory[u]* is changed by *NoteThatHTLCFulfilledOnChainByOtherUser* or *ReceiveHTLCPreimage*, by definition of these actions, the received preimage must be for an outgoing HTLC in channel c. As channels c' and c are different channels, the HTLC for which the preimage is received must be an

incoming HTLC in channel c'. The change to the variable *UserPreimageInventory[u]* in step $m_{c'}(s) \rightarrow m_{c'}(t)$ that a preimage is received is described by the action *ReceivePreimageForHTLC* of the module *ChannelEnvironment*.

4. If $s \rightarrow t$ is a step of *RequestInvoice* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The action *RequestInvoice* is not specific to a channel but only specific to user u. If user u is in channel c', step $m_{c'}(s) \rightarrow m_{c'}(t)$ is also described by *RequestInvoice* of the module *HTLCUser*. If user u is not in channel c', the global variable *Messages* might be changed in step $m_{c'}(s) \rightarrow m_{c'}(t)$ if a message is sent from user u to a user of channel c'. Such a step $m_{c'}(s) \rightarrow m_{c'}(t)$ is described by the action *RequestInvoice* of the module *ChannelEnvironment*. If no user of channel c' is the recipient of the message sent by u, then, by definition of $m_{c'}$, the variable *Messages* is unchanged in $m_{c'}(s) \rightarrow m_{c'}(t)$ and $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step.

5. If $s \rightarrow t$ is a step of *GenerateAndSendPaymentHash* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The action *GenerateAndSendPaymentHash* is not specific to a channel but only specific to user u. If user u is in channel c', step $m_{c'}(s) \rightarrow m_{c'}(t)$ is also described by *GenerateAndSendPaymentHash* of the module *HTLCUser*. If user u is not in channel c', the global variable *Messages* might be changed in step $m_{c'}(s) \rightarrow m_{c'}(t)$ if user u replies to a message sent from a user of channel c'. Such a step $m_{c'}(s) \rightarrow m_{c'}(t)$ is described by the action *GenerateAndSendPaymentHash* of the module *ChannelEnvironment*. The value included in the reply is deterministic as it can be deducted from the payment id for which an invoice is requested. Thus, the action *GenerateAndSendPaymentHash* of the module *ChannelEnvironment* defines the reply that an actual user would send. If user u replies to a message that was not sent by a user from channel c', by definition of $m_{c'}$, the variable *Messages* is unchanged in $m_{c'}(s) \rightarrow m_{c'}(t)$ and $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step.

6. If $s \rightarrow t$ is a step of *ReceivePaymentHash* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The action *ReceivePaymentHash* is not specific to a channel but only specific to user u. If user u is in channel c', step $m_{c'}(s) \rightarrow m_{c'}(t)$ is also described by *ReceivePaymentHash* of the module *HTLCUser*. If user u is not in channel c', the global variable *Messages* might be changed in step $m_{c'}(s) \rightarrow m_{c'}(t)$ if user u receives a message sent by a user from channel c'. Such a step $m_{c'}(s) \rightarrow m_{c'}(t)$ is described by the action *ReceivePaymentHash* of the module *ChannelEnvironment* that describes for the user in channel c' that the received message is removed from the *Messages* variable. If user u receives a message that was not sent by a user from channel c', by definition of $m_{c'}$, the variable *Messages* is unchanged in $m_{c'}(s) \rightarrow m_{c'}(t)$ and $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step.

7. If $s \rightarrow t$ is a step of *IgnoreInvoiceRequest* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The action *IgnoreInvoiceRequest* is not specific to a channel but only specific to user u. If user u is in channel c', step $m_{c'}(s) \rightarrow m_{c'}(t)$ is also described by *IgnoreInvoiceRequest* of the module *HTLCUser*. If user u is not in channel c', the global variable *Messages* might be changed in step $m_{c'}(s) \rightarrow m_{c'}(t)$ if user u ignores a message sent by a user from channel c'. Such a step $m_{c'}(s) \rightarrow m_{c'}(t)$ is described by the action *IgnoreMessageDuringClosing* of the module *ChannelEnvironment* that describes for the user in channel c' that the sent message is removed from the *Messages* variable. If user u ignores a message that was not sent by a user from channel c', by definition of $m_{c'}$, the variable *Messages* is unchanged in $m_{c'}(s) \rightarrow m_{c'}(t)$ and $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step.

8. If $s \rightarrow t$ is a step of *AddAndSendOutgoingHTLC* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The only variable that is changed by *AddAndSendOutgoingHTLC* and is not specific to channel c is *UserNewPayments[u]*. This variable is not changed by m_c. If user u is not part of channel c', the variable *UserNewPayments[u]* is unchanged in step $m_{c'}(s) \rightarrow m_{c'}(t)$ because this variable is unchanged in step $s \rightarrow t$. If user u is part of channel c', the change to the variable *UserNewPayments[u]* is that a payment is removed. This change is described by the action *AddOutgoingHTLCToOtherChannel* of the module *ChannelEnvironment*.

9. If $s \rightarrow t$ is a step of *ReceiveUpdateAddHTLC* of user u in channel c, then $m_{c'}(s) \rightarrow m_{c'}(t)$ is either a stuttering step, a step of an action of *ChannelEnvironment* or of *HTLCUser* for user u.

   Proof: The only variable that is changed by *ReceiveUpdateAddHTLC* and is not specific to channel c is *UserNewPayments[u]*. If user u is not part of channel c', the variable *UserNewPayments[u]* is unchanged in step $m_{c'}(s) \rightarrow m_{c'}(t)$ because this variable is unchanged in step $s \rightarrow t$. If user u is part of channel c', the change to the variable *UserNewPayments[u]* is that a payment is added that should be forwarded. This change is described by the action *AddNewForwardedPayment* of the module *ChannelEnvironment*. The action *AddNewForwardedPayment* describes the payments that can be added based on the initial payments of other users and calculates the parameters of the payment to be forwarded based on the position of the channel c' in the path of the payment.

10. Q.E.D.

    Proof: The steps above discussed all actions that change variables that have an effect on the variables in step $m_{c'}(s) \rightarrow m_{c'}(t)$. Thus, for all steps $s \rightarrow t$ of other actions, the step $m_{c'}(s) \rightarrow m_{c'}(t)$ is a stuttering step.

A.3.11. Proof of Lemma 8

Before we prove Lemma 8, we introduce and prove Lemma 21 that we need in the proof of Lemma 8.

Lemma 21.

$$\textit{SpecificationI}^{\textit{time, single}} \Rightarrow \forall x \in \mathcal{X} : relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$$

Proof: Because $\textit{SpecificationI}^{\textit{time, single}}$ corresponds largely to $\textit{SpecificationI}^{\textit{time}}$, Lemma 21 is proven analogously to the proof of Lemma 6 in Section A.3.7. We do not repeat the proof here but write only what needs to be changed to the prove of Lemma 6 to apply that prove here. The main difference between $\textit{SpecificationI}^{\textit{time}}$ and $\textit{SpecificationI}^{\textit{time, single}}$ is that $\textit{SpecificationI}^{\textit{time, single}}$ contains the module *ChannelEnvironment* (short: $CE(c,u)$).
The module *ChannelEnvironment* defines $CE(c,u)!TimelockRegions$ which is included in the definition of $relZTP^{LedgerTime}$ in $\textit{SpecificationI}^{\textit{time, single}}$. $CE(c,u)!TimelockRegions$ is defined as the union of two subsets. The first subset $CE(c,u)!T_1$ contains the absolute timelock of each incoming HTLC that has not been fulfilled. The second subset $CE(c,u)!T_2$ contains for each payment that starts at a user who is not part of channel c the timelock and the timelock +*FORWARD_DELTA* + 1 for each HTLC on the payment's route.
Because of the module *ChannelEnvironment*, $\textit{SpecificationI}^{\textit{time, single}}$ contains a further action that adds a payment to *UserNewPayments[u]* which is the action *AddNewForwardedPayment* of the module *ChannelEnvironment*. Thereby, the proof step to show that $\forall c \in C, u \in U : H(c,u)!T_1' \subseteq relZTP^{LedgerTime}$ needs to be complemented by considering steps of *AddNewForwardedPayment*. For steps of *AddNewForwardedPayment* it holds that $H(c,u)!T_1' \subseteq CE(c,u)!T_2$ because each payment that is added by the action *AddNewForwardedPayment* is already considered in $CE(c,u)!T_2$.
Further, it needs to be proven that $CE(c,u)!T_1' \subseteq CE(c,u)!T_1$ and $CE(c,u)!T_2' \subseteq CE(c,u)!T_2$. It holds that $CE(c,u)!T_1' \subseteq CE(c,u)!T_1$ because each HTLC that has not been fulfilled after a step has already been not fulfilled before the step. It holds that $CE(c,u)!T_2' \subseteq CE(c,u)!T_2$ because the definition of the set contains mostly of constants. The only exception to that is that a payment is excluded from being considered for $CE(c,u)!T_2$ once there exists an HTLC for that payment. Because HTLCs are never removed, each payment that is excluded stays excluded and, thus, it holds that $CE(c,u)!T_2' \subseteq CE(c,u)!T_2$.

We recall Lemma 8 which was defined on page 142:

Lemma 8. *For each channel $c \in C$, the mapping m_c maps each behavior $\sigma = \langle \sigma_0, \sigma_1, \sigma_2, ... \rangle$ of $\textit{SpecificationI}^{\textit{time}}$ to a behavior $\langle m_c(\sigma_0), m_c(\sigma_1), m_c(\sigma_2), ... \rangle$ of $\textit{SpecificationI}^{\textit{time, single}}$ of channel c.*

We prove the lemma by induction:
1. For all initial states of $\textit{SpecificationI}^{\textit{time}}$ and all channels $c \in C$ it holds that $m_c(s)$ is an initial state of $\textit{SpecificationI}^{\textit{time, single}}$

By definition of the initial states, applying the selection by m_c to an initial state of *SpecificationI*^{time} results in an initial state of *SpecificationI*$^{time,\ single}$.

2. Assume: $s \in \Sigma_{I,time}$ and $m_c(s) \in \Sigma_{I,time,single}$ and $s \longrightarrow t$ is a step of *SpecificationI*^{time}
   Prove: $m_c(s) \longrightarrow m_c(t)$ is a step of *SpecificationI*$^{time,\ single}$

   2.1. A step of *SpecificationI*^{time} is either a step of a (1) *PaymentChannelUser*, (2) *HTLCUser*, (3) *LedgerTime*, or (4) a final withdraw step.

   Proof: By definition of *NextII*.

   2.2. Case: $s \longrightarrow t$ is a step of *PaymentChannelUser*

   If the step $s \longrightarrow t$ is a step of *PaymentChannelUser*, it is either (1.1) a step of an action of *PaymentChannelUser* for channel c or (1.2) for another channel.

   2.2.1. Case: $s \longrightarrow t$ is a step of *PaymentChannelUser* for channel c

   To show that $m_c(s) \longrightarrow m_c(t)$ is a step of *SpecificationI*$^{time,\ single}$, we distinguish between the channel-specific variables, user-specific variables and global variables.

   2.2.1.1. The update of the channel-specific variables for channel c in step $m_c(s) \longrightarrow m_c(t)$ is a step of the module *PaymentChannelUser* in *SpecificationI*$^{time,\ single}$.

   The channel-specific variables are unchanged by the mapping m_c. The statement follows because the step $s \longrightarrow t$ is a step of the module *PaymentChannelUser* and the same module is used in *SpecificationI*$^{time,\ single}$.

   2.2.1.2. The update of the user-specific variables for user u in step $m_c(s) \longrightarrow m_c(t)$ is a step of the module *PaymentChannelUser* in *SpecificationI*$^{time,\ single}$.

   While some user-specific variables of the module *PaymentChannelUser* are unchanged by the mapping m_c, other variables (*UserPreimageInventory[u]*, *UserExtBalance[u]*, *UserChannelBalances[u]*, and *UserPayments[u]*) are changed. We show that the statement holds even with the changes of m_c to the changed variables.

   For *UserPreimageInventory[u]* the change of m_c is to remove all preimages that are not in the set of relevant preimages $R_{u,c}$. By Fig. A.3, the set of relevant preimages $R_{u,c}$ contains the preimages for all HTLCs stored in *ChannelUserVars[c][u]*. All usages of *UserPreimageInventory[u]* in module *PaymentChannelUser* consider either whether the preimage for an HTLC is in *UserPreimageInventory[u]* or whether a preimage in *UserPreimageInventory[u]* can be used for spending a transaction output. For both, only the preimages that are in the set $R_{u,c}$ are relevant because a transaction output in Lightning is only spendable with a preimage if a private key is known. Because the private keys are different for every channel, a preimage can only be used to spend a transaction by an action for a channel in which an HTLC exists to which the preimage belongs. Thus, a step of the module *PaymentChannelUser* is also possible with the values of *UserPreimageInventory[u]* reduced by the mapping m_c.

   Actions of the module *PaymentChannelUser* update the state of payments in the variable *UserPayments[u]*. All steps of these actions are also possible if the variable *UserPayments[u]* contains fewer entries.

   The opening of a channel is only initiated by the channel's funder if the funder has at least the funding amount for the channel. By an explicitly stated

assumption in *BaseSpec*, the specification ensures that this is fulfilled. Thus, the conditions in case that m_c maps *UserExtBalance[u]* to the funding amount are fulfilled in the step $m_c(s) \rightarrow m_c(t)$. The value of the variable *UserExtBalance[u]* is only updated when a funding transaction is published and, thereby, the corresponding channel is opened. In such a step, the mapping of m_c changes to map *UserExtBalance[u]* to 0 which corresponds to removing the funding amount from the user's external balance which corresponds to the change in step $s \rightarrow t$. In the definition of the module *PaymentChannelUser*, only the field for channel c of the variable *UserChannelBalances[u]* is accessed. Because the field for channel c is not changed by the mapping m_c, the step $m_c(s) \rightarrow m_c(t)$ is also valid if the mapping stored in the variable *UserChannelBalances[u]* is reduced to an entry for channel c.

Thus, a step of the module *PaymentChannelUser* is also possible with the values of *UserPreimageInventory[u]*, *UserExtBalance[u]*, *UserChannelBalances[u]*, and *UserPayments[u]* reduced by the mapping m_c.

2.2.1.3. The update of the global variables in step $m_c(s) \rightarrow m_c(t)$ is a step of the module *PaymentChannelUser* in *SpecificationI*$^{time,\ single}$.

Two global variables are used by the module *PaymentChannelUser* and both are changed by the mapping m_c: *LedgerTx* and *LedgerTime*. The variable *LedgerTx* is reduced to all transactions that are related to channel c. Other transactions do not affect whether an action of module *PaymentChannelUser* are enabled.

For steps of the module *PaymentChannelUser* in *SpecificationI*$^{time,\ single}$, it must hold that the variable *LedgerTime* is unchanged. The value of $m_c(t).LedgerTime$ is defined as the maximum of $m_c(s).LedgerTime$ and all values in $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state t that are lower than or equal to $t.LedgerTime$. We must show that the value of the variable *LedgerTime* is unchanged, i.e. that $m_c(s).LedgerTime = m_c(t).LedgerTime$. This holds if the largest value of $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state t that is lower than or equal to $t.LedgerTime$ is lower than or equal to the largest value of $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state s that is lower than or equal to $s.LedgerTime$. This holds if it holds in *SpecificationI*$^{time,\ single}$ that $relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$ which is shown by Lemma 21.

2.2.1.4. Q.E.D.

Because the steps 2.2.1.1, 2.2.1.2, 2.2.1.3 cover all variables of *SpecificationI*$^{time,\ single}$.

2.2.2. Case: $s \rightarrow t$ is a step of *PaymentChannelUser* for a channel $c' \neq c$

The variables in *SpecificationI*^{time} that are specific to channel c' do not change between states s and t. However, the variables for the users participating in channel c and channel c' might change in the step $s \rightarrow t$ or global variables might change. For example, a user of channel c and channel c' might receive a preimage. To account for such changes, *SpecificationI*$^{time,\ single}$ contains the module *ChannelEnvironment*. The module *ChannelEnvironment* abstracts all actions that can happen in other payment channels. By Lemma 7, it follows that $m_c(s) \rightarrow m_c(t)$ is a step of *SpecificationI*$^{time,\ single}$.

2.2.3. Q.E.D.

By 2.2.1 and 2.2.2.

2.3. Case: $s \rightarrow t$ is a step of *HTLCUser*

A step of *HTLCUser* is either a step of an action of *HTLCUser* for channel c or for another channel.

2.3.1. Case: $s \rightarrow t$ is a step of *HTLCUser* for channel c

To show that $m_c(s) \rightarrow m_c(t)$ is a step of $\textit{SpecificationI}^{\textit{time, single}}$, we distinguish between the channel-specific variables, user-specific variables and global variables.

2.3.1.1. The update of the channel-specific variables for channel c in step $m_c(s) \rightarrow m_c(t)$ is a step of the module *HTLCUser* in $\textit{SpecificationI}^{\textit{time, single}}$.

The channel-specific variables are unchanged by the mapping m_c. The statement follows because the step $s \rightarrow t$ is a step of the module *HTLCUser* and the same module is used in $\textit{SpecificationI}^{\textit{time, single}}$.

2.3.1.2. The update of the user-specific variables for user u in step $m_c(s) \rightarrow m_c(t)$ is a step of the module *HTLCUser* in $\textit{SpecificationI}^{\textit{time, single}}$.

While some user-specific variables are unchanged by the mapping m_c, other variables used by the module *HTLCUser* (*UserPreimageInventory[u]*, *UserChannelBalances[u]*, and *UserPayments[u]*) are changed. We show that the statement holds even with the changes of m_c to the changed variables.

For *UserPreimageInventory[u]* the change of m_c is to remove all preimages that are not in the set of relevant preimages $R_{u,c}$. By Fig. A.3, the set of relevant preimages $R_{u,c}$ contains the preimages for all HTLCs stored in *ChannelUserVars[c][u]*. All usages of *UserPreimageInventory[u]* in module *HTLCUser* consider only preimages for HTLCs in channel c. Therefore, only the preimages that are also in the set $R_{u,c}$ are relevant.

Actions of the module *PaymentChannelUser* use the variable *UserPayments[u]* to update the state of payments in the variable *UserPayments[u]*. All steps of these actions are also possible if the variable *UserPayments[u]* contains fewer entries. Further, the action *ReceiveUpdateAddHTLC* checks whether the value of a received payment match the values stored in *UserPayments[u]*. If there exists a payment for which user u is the payment's receiver in s.*UserPayments[u]* and user u makes a *ReceiveUpdateAddHTLC* step, it holds that the other user in channel is the second to last hop on the payment's route. Thus, the payment is also contained in $m_c(s)$.*UserPayments[u]*.

In the definition of the module *HTLCUser*, only the field for channel c of the variable *UserChannelBalances[u]* is accessed. Because the field for channel c is not changed by the mapping m_c, the step $m_c(s) \rightarrow m_c(t)$ is also valid if the mapping stored in the variable *UserChannelBalances[u]* is reduced to an entry for channel c.

Thus, a step of the module *HTLCUser* is also possible with the values of *UserPreimageInventory[u]*, *UserChannelBalances[u]*, and *UserPayments[u]* reduced by the mapping m_c.

2.3.1.3. The update of the global variables in step $m_c(s) \rightarrow m_c(t)$ is a step of the module *HTLCUser* in *SpecificationI*$^{time,\ single}$.

Two global variables are used by the module *HTLCUser* and both are changed by the mapping m_c: *Messages* and *LedgerTime*. The variable *Messages* is reduced to all messages that are sent from or to a user who is part of channel c. Messages with different senders and receivers do not affect steps of the module *HTLCUser*. Analogously to the module *PaymentChannelUser*, it must hold for steps of the module *HTLCUser* in *SpecificationI*$^{time,\ single}$ that the variable *LedgerTime* is unchanged. The value of $m_c(t).LedgerTime$ is defined as the maximum of $m_c(s).LedgerTime$ and all values in $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state t that are lower than or equal to $t.LedgerTime$. We must show that the value of the variable *LedgerTime* is unchanged, i.e. that $m_c(s).LedgerTime = m_c(t).LedgerTime$. This holds if the largest value of $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state t that is lower than or equal to $t.LedgerTime$ is lower than or equal to the largest value of $relZTP^{LedgerTime}$ of *SpecificationI*$^{time,\ single}$ in state s that is lower than or equal to $s.LedgerTime$. This holds if it holds in *SpecificationI*$^{time,\ single}$ that $relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$ which is shown by Lemma 21.

2.3.1.4. Q.E.D.

Because the steps 2.3.1.1, 2.3.1.2, 2.3.1.3 cover all variables of *SpecificationI*$^{time,\ single}$.

2.3.2. Case: $s \rightarrow t$ is a step of *HTLCUser* for a channel $c' \neq c$

The variables in *SpecificationI*^{time} specifically for channel c' do not change between states s and t. However, the variables for the users participating in channel c and c' might change in the step $s \rightarrow t$ or global variables might change. As in step 2.2, we use the module *ChannelEnvironment* to account for this. For every action in *HTLCUser* that changes a variable that is also used in another channel, there is an action in *ChannelEnvironment* that describes the changes to the variables. By Lemma 7, it follows that $m_c(s) \rightarrow m_c(t)$ is a step of *SpecificationI*$^{time,\ single}$.

2.3.3. Q.E.D.

By 2.3.1 and 2.3.2.

2.4. Case: $s \rightarrow t$ is an *AdvanceLedgerTime* step

The only change between states s and t is an increase of clocks, i.e. *LedgerTime* and the clocks in *TxAge*. By the definition of m_c, the value of *LedgerTime* can only increase in step $m_c(s) \rightarrow m_c(t)$ and *LedgerTime* in state $m_c(t)$ is set to a value that is in $relZTP^{LedgerTime}$. Because the time bounds in *SpecificationI*$^{time,\ single}$ are, by definition, a subset of the time bounds in *SpecificationI*^{time}, the step $m_c(s) \rightarrow m_c(t)$ complies with the time bounds of *SpecificationI*$^{time,\ single}$. The clocks in *TxAge* are unchanged by the mapping m_c. Because all other variables are unchanged in the step $s \rightarrow t$ and the mapping m_c does not depend on the value of a clock, these variables are also unchanged in the step $m_c(s) \rightarrow m_c(t)$. It follows that the step $m_c(s) \rightarrow m_c(t)$ is a step of *SpecificationI*$^{time,\ single}$.

2.5. Case: $s \rightarrow t$ is a final withdraw step of action *WithdrawBalancesAfterChannelClosed*

The final withdraw action *WithdrawBalancesAfterChannelClosed* is defined in *SpecificationI*$^{time,\ single}$ as in *SpecificationI*^{time} with the exception of the new value of *UserExtBalance[u]* for each user u. Thus, we need to show that the change of *UserExtBalance* in $m_c(s) \rightarrow m_c(t)$ conforms to the final withdraw action of *SpecificationI*$^{time,\ single}$. The balance of dishonest users does not change in step $s \rightarrow t$ which matches the definition of the final withdraw action in *SpecificationI*$^{time,\ single}$. In step $s \rightarrow t$, the value of *UserExtBalance[u]* for each honest user u is increased by user u's on-chain balance. In *SpecificationI*$^{time,\ single}$ the visible on-chain balance might be less than in *SpecificationI*^{time} because only the on-chain transactions for channel c are visible. To account for the balances in other channels, the action *WithdrawBalancesAfterChannelClosed* specifies that the value of *UserExtBalance[u]* for each honest user u must be increased by the visible on-chain balance and the user u's balance in other channels. Because this sum equals a user's complete balance as defined in *SpecificationI*^{time}, the update of the variable *UserExtBalance[u]* in step $m_c(s) \rightarrow m_c(t)$ matches the definition of *WithdrawBalancesAfterChannelClosed* in *SpecificationI*$^{time,\ single}$.

The other changes of variables are equal in the definitions of *SpecificationI*^{time} and *SpecificationI*$^{time,\ single}$ and the changed variables are unchanged by the mapping m_c with exception of the variable *LedgerTime*. The variable *LedgerTime* is increased to the final value *TERMINATION* in both definitions. This value is unchanged by the mapping m_c. Thus, while the value of the variable *LedgerTime* in the states s and $m_c(s)$ might be different, the value in the states t and $m_c(t)$ equals *TERMINATION*. Because the value of *LedgerTime* in state s is only checked to be less than *TERMINATION* in the definition of the action *WithdrawBalancesAfterChannelClosed*, a difference of the value of *LedgerTime* between the states s and $m_c(s)$ is not relevant.

Some variables that the action *WithdrawBalancesAfterChannelClosed* refers to are unchanged by the mapping m_c. Variables that the action depends on and that are changed by m_c are *ChannelMessages*, *LedgerTx*, *UserExtBalance[u]*, and *UserChannelBalances[u]* and the constant *ActiveChannels*. The change of *LedgerTx* has already been discussed above. The action *WithdrawBalancesAfterChannelClosed* checks for each channel that there is no `update_fulfill_htlc` message in *ChannelMessages[c]*. Because the mapping m_c only removes messages from *ChannelMessages[u]*, this condition is fulfilled in step $m_c(s) \rightarrow m_c(t)$ if it is fulfilled in $s \rightarrow t$. The same holds for the constant *ActiveChannels* which is only a single channel in *SpecificationI*$^{time,\ single}$ compared to all channels in *SpecificationII*. The action *WithdrawBalancesAfterChannelClosed* contains conditions that are universally quantified over *ActiveChannels*. Because the value of *ActiveChannels* in *SpecificationI*$^{time,\ single}$ is contained in the set *ActiveChannels* in *SpecificationI*^{time}, it holds that these conditions are fulfilled in step $m_c(s) \rightarrow m_c(t)$ if they are fulfilled in $s \rightarrow t$. During a step of the action *WithdrawBalancesAfterChannelClosed*, the mapping of m_c changes to map *UserExtBalance[u]* to a user's on-chain balance related to channel c which corresponds to the definition of the action *WithdrawBalancesAfterChannelClosed*.

The definition of the action *WithdrawBalancesAfterChannelClosed* accesses only the field for channel c of the variable *UserChannelBalances[u]*. Because the field for channel c is not changed by the mapping m_c, the step $m_c(s) \rightarrow m_c(t)$ is also valid if

the mapping stored in the variable *UserChannelBalances[u]* is reduced to an entry for channel c.
We conclude that the step $m_c(s) \rightarrow m_c(t)$ is a step of action *WithdrawBalancesAfterChannelClosed* of *SpecificationI*$^{time,\ single}$.

2.6. Q.E.D.

By 2.2, 2.3, 2.4, and 2.5 because a step in *SpecificationI*^{time} can only be either a step of *PaymentChannelUser*, *HTLCUser*, LedgerTime, or a final withdraw step.

3. Q.E.D.

By induction using 1 and 2.

A.3.12. Proof of Lemma 10

We recall Lemma 10 which was defined on page 148:

Lemma 10. *A step of AbstractEnforcement or HTLCUser in SpecificationII*^{single} *is also possible if the variables that are reduced by* m_c *and are used in SpecificationII*^{single} *(i.e., UserPreimageInventory[u], UserLatePreimages[u], UserNewPayments[u], UserPayments[u], UserExtBalance[u], UserChannelBalances[u], Messages) contain additional values that are filtered out by* m_c *(see Definition 19).*

Proof:

1. All conditions on the first state (unprimed values) of a step of *AbstractEnforcement* or *HTLCUser* are valid even if the variables that are reduced by m_c contain additional values that are filtered out by m_c.

   Proof: We show that this property holds for every action of *AbstractEnforcement* and *HTLCUser* and every variable that is reduced by m_c. In the following, we discuss all relevant cases. The remaining cases are either trivial or analogous to the discussed cases.
   The action *GenerateAndSendPaymentHash* of *HTLCUser* adds a preimage for the requested payment to *UserPreimageInventory[u]* and a payment secret to *UserPaymentSecretForPreimage[u]*. The action contains a condition that the generated preimage may not already be in *UserPreimageInventory[u]*. Because the value chosen for the preimage in the formalization depends on and is unique for the payment that the preimage is for, if the preimage has already been generated then it must have been generated by this action and must have been added to the domain of *UserPaymentSecretForPreimage[u]* and, therefore, it is not filtered out by m_c.
   In *AbstractEnforcement*, some actions have checks whether the preimage of an HTLC is in *UserPreimageInventory[u]*. Because these checks are only for preimages for HTLCs of the users and m_c retains all preimages for which an HTLC exists, these checks return the same result if *UserPreimageInventory[u]* contains more values.
   The values of the variables *UserPayments[u]* and *UserChannelBalances[u]* are only used to set new values for these variables which is discussed in the proof step.

The value of the variable *UserExtBalance[u]* is used only once in a condition that ensures that a user's external balance is greater than or equal to the user's funding amount of a channel. If the value of the variable *UserExtBalance[u]* is greater, this condition is still fulfilled.

2. All conditions on the second state (primed values) of a step of *AbstractEnforcement* or *HTLCUser* are valid even if the variables that are reduced by m_c contain additional values that are filtered out by m_c.

   Proof: The values of the concerned variables in the second state of a step are always described by the actions of *AbstractEnforcement* and *HTLCUser* by describing a change and not directly the new value. More concretely, the actions define that a value is added or removed to the set and all remaining values of the set remain unchanged. Thus, a step of *AbstractEnforcement* or *HTLCUser* is still valid if the concerned variables contain additional values.

3. Q.E.D.

   Proof: Because the conditions on the first and the second state are valid, a step of *AbstractEnforcement* or *HTLCUser* is also a valid step if the variables that are reduced by m_c contain additional values that are filtered out by m_c.

A.3.13. Proof of Theorem 3

Before we prove Theorem 3, we introduce a lemma that we use during the proof of Theorem 3:

Lemma 22. *The refinement mapping f maps steps of PaymentChannelUser to steps of AbstractEnforcement, steps of HTLCUser to steps of HTLCUser, steps of ChannelEnvironment to steps of ChannelEnvironment, and steps of LedgerTime to steps of LedgerTime or stuttering steps.*

Proof: By design of the refinement mapping f, steps of an action A of *HTLCUser* and *ChannelEnvironment* are mapped to steps of the same action A in *SpecificationII*^{single}. This mapping of steps to steps of the same action is implemented in the refinement mapping by leaving the variables or the fields of variables unchanged by the refinement mapping that are updated by the actions of *HTLCUser* and *ChannelEnvironment*. Because the values of the refinement mapping of all variables except LedgerTime do not depend on LedgerTime, these values are unchanged, if the value of LedgerTime changes. Because the refinement mapping does not change the value of the variable *LedgerTime* and no action in another module then the module *LedgerTime* describes a change of the variable *LedgerTime*, a step of *LedgerTime* is mapped to a step of *LedgerTime*. Steps of the module *PaymentChannelUser* are mapped to steps of the module *AbstractEnforcement*.

We recall Theorem 3 which was defined on page 150:

Theorem 3. *The mapping g is a refinement mapping from SpecificationI*^{time} *to SpecificationII.*

1. For all s in the state space of *SpecificationI*^{time} it holds that the externally visible variables of state $g(s)$) equal those of state s

   Proof: The externally visible variables are for all users $u \in U$: *UserPayments[u]*, *UserExtBalance[u]*, *UserChannelBalances[u]*, and *UserHonest[u]*. These variables are by definition of g unchanged by g.

2. For all initial states s of *SpecificationI*^{time}, $g(s)$ is an initial state of *SpecificationII*

   Proof: All variables in state s are mapped by g either by directly assigning the value of the variable in state s or by applying the refinement mapping f. By definition of f, f does not change values in the initial states. By definition of *Init* of *SpecificationI*^{time} and *SpecificationII*, it follows that all variables of a state of *SpecificationI*^{time} are mapped to values of an initial state of *SpecificationII*.

3. For all steps $s \rightarrow t$ of *SpecificationI*^{time}, $g(s) \rightarrow g(t)$ is a step of *SpecificationII*.

   Proof:

   A step of *SpecificationI*^{time} can be one of the following.

   3.1. Case: $s \rightarrow t$ is a step of an action of the module *LedgerTime* in *SpecificationI*^{time}.
   Prove: $g(s) \rightarrow g(t)$ is a step of the module *LedgerTime* in *SpecificationII*.

   The only variables that are updated in the step are the variables *LedgerTime* and *TxAge*. For the step $g(s) \rightarrow g(t)$, only the variable *LedgerTime* is relevant because the variable *TxAge* does not exist in *SpecificationII*.

   3.1.1. All variables other than *LedgerTime* are unchanged in step $g(s) \rightarrow g(t)$

   Proof: Because these variables are unchanged in the step $s \rightarrow t$ and the definition of the refinement mapping f does not depend on the value of *LedgerTime*.

   3.1.2. $g(s).\mathit{LedgerTime} = s.\mathit{LedgerTime} \land g(t).\mathit{LedgerTime} = t.\mathit{LedgerTime}$

   Proof: By definition of g.

   3.1.3. $\forall b \in B^{LedgerTime} : g(s).\mathit{LedgerTime} \leq b \Rightarrow g(t).\mathit{LedgerTime} \leq b$

   Proof: Because $s \rightarrow t$ is a step of module *LedgerTime* and 3.1.2, we only need to show that if a time bound exists in *SpecificationII* then the same time bound exists in *SpecificationI*^{time}. Because the set of time bounds in the specifications are composed as the union of time bounds of all channels, it suffices to show for each channel that each time bound that exists in *SpecificationII* exists also in *SpecificationI*^{time}. By definition of the operator *TimeBounds* in module *PaymentChannelUser*, the value of *TimeBounds* is the same for a state s of *SpecificationI*^{time} and the state $m_c(s)$ of *SpecificationI*$^{time,\ single}$. By Lemma 9 and Lemma 22 each step of module *LedgerTime* in *SpecificationI*$^{time,\ single}$ is mapped to a step of module *LedgerTime* in *SpecificationII*^{single}. Thus, each increase in *LedgerTime* that is possible in a channel in *SpecificationII* is also possible in *SpecificationI*^{time}.

   3.1.4. Q.E.D.

   Proof: By 3.1.1, 3.1.2, and 3.1.3.

   3.2. Case: $s \rightarrow t$ is a step of an action of the modules *HTLCUser* or *PaymentChannelUser* in *SpecificationI*^{time}.
   Prove: $g(s) \rightarrow g(t)$ is a step of the modules *HTLCUser* or *AbstractEnforcement* in *SpecificationII*.

Let $c \in C$ be the channel and u the user for which the step $s \to t$ is a step of an action of *HTLCUser* or *PaymentChannelUser*.

3.2.1. The step $m_c(s) \to m_c(t)$ is a step of an action of *HTLCUser* or *PaymentChannelUser* for user u in the instance of $\mathit{SpecificationI}^{\mathit{time,\ single}}$ that is described by the mapping m_c

By Lemma 8.

3.2.2. The step $f(m_c(s)) \to f(m_c(t))$ is a step of *HTLCUser* or *AbstractEnforcement* in $\mathit{SpecificationII}^{\mathit{single}}$.

By Lemma 9 and Lemma 22.

3.2.3. The channel-specific variables of all channels $c' \neq c$ and the user-specific variables of all users $u' \neq u$ are unchanged in the step $s \to t$ and these variables are also unchanged in step $g(s) \to g(t)$.

Because the step $s \to t$ is a step of *HTLCUser* or *PaymentChannelUser* and by definition of g.

3.2.4. Only global variables, user-specific variables for user u and channel-specific variables for channel c can be changed in the step $g(s) \to g(t)$.

By 3.2.3.

3.2.5. The updates of channel-specific variables for channel c in the step $g(s) \to g(t)$ conform to an action of *HTLCUser* or *AbstractEnforcement*.

The updates of channel-specific variables for channel c are the same the step $g(s) \to g(t)$ as in the step $f(m_c(s)) \to f(m_c(t))$ by definition of g. As shown in 3.2.2, the updates of these variables conform to an action of the modules *HTLCUser* or *AbstractEnforcement*.

3.2.6. The updates of user-specific variables for channel c in the step $g(s) \to g(t)$ conform to an action of *HTLCUser* or *AbstractEnforcement*.

The updates of user-specific variables for user u are the same in the step $g(s) \to g(t)$ as in the step $f(m_c(s)) \to f(m_c(t))$ except that they might contain additional values of other channels which were removed by the mapping m_c. As shown in Lemma 10, these additional values in the step $g(s) \to g(t)$ do not affect that the updates of these variables in the step $g(s) \to g(t)$ conform to an action of *HTLCUser* or *AbstractEnforcement*.

3.2.7. The updates of global variables in the step $g(s) \to g(t)$ conform to an action of *HTLCUser* or *AbstractEnforcement*.

There are the two global variables *Messages* and *LedgerTime* in *SpecificationII*.
For the variable *Messages*, the case is analogous to 3.2.6: There can be additional values in the step $g(s) \to g(t)$ compared to the step $f(m_c(s)) \to f(m_c(t))$. As shown in Lemma 10, these additional values in the step $g(s) \to g(t)$ do not affect that the updates of these variables in the step $g(s) \to g(t)$ conform to an action of *HTLCUser* or *AbstractEnforcement*.
For the variable *LedgerTime*, the value in the step $g(s) \to g(t)$ might be larger as the value of *LedgerTime* in the step $f(m_c(s)) \to f(m_c(t))$ because the mapping m_c might reduce the value of *LedgerTime*. However, there can be states s' and t'

that are equal to states s and t but in the states s' and t' there is a payment that is not related to channel c but the payment's timelock equals $s.LedgerTime$ and is included in *ChannelEnvironment*$(c, u)!TimelockRegions$. In this case, the value of *LedgerTime* in state $m_c(s')$ is equal to the value of *LedgerTime* in state s' and, thus, equal to the value of *LedgerTime* in state s. Thus, the value of *LedgerTime* is equal in the steps $f(m_c(s')) \rightarrow f(m_c(t'))$ and $g(s) \rightarrow g(t)$. Thus, the value of *LedgerTime* in the step $g(s) \rightarrow g(t)$ conforms to an action of *HTLCUser* or *AbstractEnforcement*.

3.2.8. Q.E.D.

By 3.2.3, 3.2.5, 3.2.6, and 3.2.7.

3.3. Case: $s \rightarrow t$ is a step of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationI*^{time}.

Prove: $g(s) \rightarrow g(t)$ is a step of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationII*.

The definitions of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationI*^{time} and *SpecificationII* are specified similarly but differ in some conditions. Most conditions of the definition of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationI*^{time} directly imply the corresponding condition in *SpecificationII*. The only exception is the definition of the new value of the variable *UserExtBalance[u]*. In *SpecificationI*^{time}, this variable is assigned, for each honest user u, the value of the sum of all outputs on the blockchain that are spendable by user u. The values of *UserExtBalance[u]* for dishonest users u are unchanged because these values or not relevant for the security property, i.e., the action models that only honest users withdraw their balance. In *SpecificationII*, however, the blockchain is abstracted and the new value of the variable *UserExtBalance[u]* needs to be defined differently. If a channel is closed honestly, a user's new external balance is increased by the sum of a user's balances in all closed channels. If a channel is closed dishonestly, the honest user retrieves the whole capacity of the channel. Thus, the specification of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationII* defines that an honest user u retrieves the sum of a user's balances in all channels and, optionally, the counterparty's balance in all channels where the counterparty has cheated.
By Lemma 9, the final withdraw steps of each channel in *SpecificationI*$^{time,\ single}$ are mapped to final withdraw steps of the channel in *SpecificationII*^{single}. Because definitions of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationI*^{time} and *SpecificationII* simply sum up the balances of all channels, this implies the statement to prove.

3.4. Q.E.D.

Proof: By 3.1, 3.2, 3.3 and because these cases cover all possible steps in *SpecificationI*^{time}.

4. Q.E.D.

Proof: By 1, 2, 3, and the definition of refinement mappings and because *SpecificationII* contains the same fairness condition as *SpecificationI*^{time} that only behaviors are valid that end in a state in which all users have withdrawn their balance.

A.3.14. Proof of Lemma 11

We recall Lemma 11 which was defined on page 157:

Lemma 11. *In SpecificationII it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T_d^y(s))$$

Proof: $B^{LedgerTime}$ is defined by *TimeBounds* of *AbstractEnforcement*. As the variable *LedgerTime* does not occur in the definition of $TimeBounds$, the definition of $TimeBounds$ is independent of *LedgerTime* and, thus, Assumption 4.1 holds for $B^{LedgerTime}$.

A.3.15. Proof of Lemma 12

We recall Lemma 12 which was defined on page 157:

Lemma 12. $relZTP^x$ *of SpecificationII*^{time} *meets Assumption 4.2, i.e.* $\forall x \in \mathcal{X}, s \in \hat{\Sigma}$: $ZTP^x(s) \subseteq relZTP^x(s)$.

The set $relZTP^{LedgerTime}$ is defined as $relZTP$ in *SpecificationII*^{time} which is defined as the union of $AC(c)!TimelockRegions$ for all channels c and $HU(c,u)!TimelockRegions$ for all channels c and their users u. The operator $AC(c)!TimelockRegions$ is defined as the set that contains, for all incoming and outgoing HTLCs of both users, the HTLC's timelock and the HTLC's timelock plus the forward delta.

In the proof, we use Definition 23 as introduced in the proof in Section 4.3.1.

Proof: The module *AbstractEnforcement* contains seven actions that depend on *LedgerTime*:

1. $ZTP^{LedgerTime}_{UpdatePaymentChannel}(s) \subseteq relZTP^{LedgerTime}(s)$

   Proof: The action *UpdatePaymentChannel* depends on the value of *LedgerTime* to choose the HTLCs to add and to remove. To this end, the condition that is used is the condition that an HTLCs timelock is greater than *LedgerTime*. Thus, the set $ZTP^{LedgerTime}_{UpdatePaymentChannel}(s)$ contains the timelock of all HTLCs that are in state OFF-TIMEDOUT because at the timelock of an HTLC a step that removes the HTLC is possible that is not possible at the directly preceding point in time. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$. By the definition of $TimelockRegions$ in the module *HTLCUser*, the value is already in the set $relZTP^{LedgerTime}(s)$ before an HTLC is directly added to the variables of a user (see proof on 324).
   The action *UpdatePaymentChannel* also contains a condition that under certain conditions the value of LedgerTime is smaller than an HTLC's absolute timelock plus the forward delta *FORWARD_DELTA*. If the value of *LedgerTime* is larger than this value, a step of *UpdatePaymentChannel* is not possible anymore. Thus, this condition does not imply that the set $ZTP^{LedgerTime}_{UpdatePaymentChannel}(s)$ contains additional values.

2. $ZTP^{LedgerTime}_{SetOnChainHTLCsAndCheater}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: As the action *UpdatePaymentChannel*, the action *SetOnChainHTLCsAndCheater* contains a condition that under certain conditions the value of *LedgerTime* is smaller than an HTLC's absolute timelock plus the forward delta. With larger values of *LedgerTime* steps are not possible anymore. Thus, this condition does not justify why a value is in the set $ZTP^{LedgerTime}_{UpdatePaymentChannel}(s)$.
   Another condition requires a user to have at least the balance of incoming HTLCs that cannot be persisted because the user was dishonest and the value of *LedgerTime* is larger than or equal to the HTLC's timelock plus the forward delta. Again, while this condition might be the reason for a step to not be possible, it does not justify why a value is in the set $ZTP^{LedgerTime}_{UpdatePaymentChannel}(s)$. We conclude that the set $ZTP^{LedgerTime}_{SetOnChainHTLCsAndCheater}(s)$ is empty.
3. $ZTP^{LedgerTime}_{FulfillIncomingHTLCsOnChain}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *FulfillIncomingHTLCsOnChain* can fulfill an HTLC on-chain as long as the HTLC's timelock plus the forward delta is greater than *LedgerTime*. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of *LedgerTime* reaches an HTLC's timelock plus the forward delta no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.
   Additionally, the action has the same condition for the balance of users as the action *SetOnChainHTLCsAndCheater* that also does not lead to steps becoming possible. Therefore, $ZTP^{LedgerTime}_{FulfillIncomingHTLCsOnChain}(s)$ is empty.
4. $ZTP^{LedgerTime}_{NoteFulfilledHTLCsOnChain}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *NoteFulfilledHTLCsOnChain* can note a fulfilled HTLC on-chain as long as the HTLC's timelock plus the forward delta is greater than *LedgerTime*. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of *LedgerTime* reaches an HTLC's timelock plus the forward delta no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.
   If the preimage for an HTLC is learned when the value of *LedgerTime* is greater than the HTLC's timelock plus the forward delta, the preimage is added to the set *UserLatePreimages[u]*. Thus, for each HTLC that can be fulfilled, the set $ZTP^{LedgerTime}_{NoteFulfilledHTLCsOnChain}(s)$ contains the HTLC's timelock plus the forward delta. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$.
5. $ZTP^{LedgerTime}_{CommitHTLCsOnChain}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *CommitHTLCsOnChain* can persist an HTLC on-chain as long as the value of *LedgerTime* is smaller than the HTLC's timelock plus the forward delta. Because the action chooses a subset of persistable HTLCs as the HTLCs to persist, if the value of *LedgerTime* reaches an HTLC's timelock plus the forward delta no new step becomes possible but only all steps that include persisting this HTLC become impossible.
   The action *CommitHTLCsOnChain* also contains a condition that under certain conditions the value of *LedgerTime* is smaller than an HTLC's absolute timelock plus

the forward delta. For larger values of *LedgerTime* steps become impossible but this condition does not allow for new steps to become possible.
If the preimage for an HTLC is persisted when the value of *LedgerTime* is greater than the HTLC's timelock plus the forward delta, the preimage is added to the set *UserLatePreimages[u]* of the user who learns the preimage. Thus, for each HTLC that can be persisted, the set $ZTP^{LedgerTime}_{CommitHTLCsOnChain}(s)$ includes the HTLC's timelock plus the forward delta. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$.
Another condition requires a user to have at least the balance of incoming HTLCs that cannot be persisted because the user was dishonest and the value of *LedgerTime* is larger than or equal to the HTLC's timelock plus the forward delta. This condition might lead to steps becoming impossible but there are no steps of *CommitHTLCsOnChain* that can become possible.

6. $ZTP^{LedgerTime}_{FulfillHTLCsOnChain}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *FulfillHTLCsOnChain* can fulfill an HTLC on-chain as long as the HTLC's timelock plus the forward delta is greater than *LedgerTime*. Because the action chooses a subset of fulfillable HTLCs as the HTLCs to fulfill, if the value of *LedgerTime* reaches an HTLC's timelock plus the forward delta no new step becomes possible but only all steps that include fulfilling this HTLC become impossible.
   If the preimage for an HTLC is learned when the value of *LedgerTime* is greater than the HTLC's timelock plus the forward delta, the preimage is added to the set *UserLatePreimages[u]*. Thus, for each HTLC that can be fulfilled, the set $ZTP^{LedgerTime}_{FulfillHTLCsOnChain}(s)$ includes the HTLC's timelock plus the forward delta. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$.
7. $ZTP^{LedgerTime}_{ClosePaymentChannel}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *ClosePaymentChannel* contains a condition in the operator *ValidMapping* that verifies that an HTLC can only be timed out if the value of *LedgerTime* is at least the HTLCs timelock. Thus, steps in which the HTLC is timed out are only valid from a state on in which the value of *LedgerTime* is at least the HTLC's timelock. Therefore, the $relZTP^{LedgerTime}(s)$ must include the HTLC's timelock. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$. The operator *ValidMapping* contains a check that an HTLC can only be persisted as long as the value of *LedgerTime* is lower than the HTLC's timelock plus the forward delta. An increasing value of *LedgerTime* does not enable new steps to become possible.
   If the preimage for an HTLC is learned when the value of *LedgerTime* is greater than the HTLC's timelock plus the forward delta, the preimage is added to the set *UserLatePreimages[u]*. Thus, for each HTLC that can be fulfilled, the set $ZTP^{LedgerTime}_{ClosePaymentChannel}(s)$ contains the HTLC's timelock plus the forward delta. This is fulfilled by the definition of *TimelockRegions* of the module *AbstractEnforcement* that is included by $relZTP^{LedgerTime}(s)$.

8. For each action A of the modules *HTLCUser* and *LedgerTime* it holds that $ZTP_A^{LedgerTime}(s) \subseteq relZTP^{LedgerTime}(s)$

   Proof: By the proof on page 326.

9. Q.E.D.

   By the steps 1, 2, 3, 4, 5, 6, 7, and 8 and we have proven for all subactions of $InnerNext$ that $\forall x \in \mathcal{X}, s \in \Sigma : ZTP^x(s) \subseteq relZTP^x(s)$.

A.3.16. Proof of Lemma 13

We recall Lemma 13 which was defined on page 157:

Lemma 13. *For any two states s, t of SpecificationII and all clocks $x \in \mathcal{X}$ (i.e., the clock LedgerTime), it holds that*

$$\textit{NextII} \Rightarrow relZTP^{x\prime} \subseteq relZTP^x$$

The following proof is analogous to the proof of Lemma 6 in Section A.3.7.

The specification defines the set $relZTP^{LedgerTime}$ as the union of the sets *TimelockRegions* of the modules *AbstractEnforcement* and the module *HTLCUser* and the set $\{t + 1 : t \in \textit{TimeBounds}\}$. These sets are again defined as the union of smaller sets.

The set $AC(c, u)!TimelockRegions$ is the union of two sets:

- A set T_1 that contains the HTLC's timelock for all HTLCs independent of their state.
- A set T_2 that contains the HTLC's timelock +*FORWARD_DELTA* + 1 for all HTLCs independent of their state.

We refer to these sets as $AC(c, u)!T_1$ to $AC(c, u)!T_2$.

The set $H(c, u)!TimelockRegions$ is a union of three sets:

- A set T_1 that contains for each payment in *NewPayments* the timelock and the timelock +*FORWARD_DELTA* + 1 for each HTLC on the payment's route.
- A set T_2 that contains the timelock of each HTLC and the timelock + *FORWARD_DELTA* + 1 for which an `update_add_htlc` message is contained in *ChannelMessages*. This set includes also the HTLCs that are contained in the message as nested payload to be forwarded.
- A set T_3 that contains the timelock and the timelock +*FORWARD_DELTA* + 1 for each HTLC that has not been fulfilled.

We refer to these sets as $H(c, u)!T_1$ to $H(c, u)!T_3$.

The set $AC(c, u)!TimeBounds$ is a union of two sets:

- A set B_1 that contains the timelock $+FORWARD\_DELTA - 1$ for certain HTLCs and, thus, the set $\{t + 1 : t \in B_1\}$ is always a subset of $AC(c,u)!TimelockRegions$.
- A set B_2 that contains the value MAX_TIME and, if the protocol state does not equal `closed`, the value $MAX\_TIME - TO\_SELF\_DELAY$.

We refer to these sets as $AC(c,u)!B_1$ and $AC(c,u)!B_2$.

In the proof, we show that for each of these sets it holds that the set in state t is a subset of the set in state s.

Proof:

1. $AdvanceTimeIII \Rightarrow relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$

   Proof: Because the definition of $relZTP^{LedgerTime}$ does not depend on the value of a clock variable but uses only non-clock variables that are unchanged by $AdvanceTimeIII$.

We look at steps that do not create a new HTLC.

2. Case: $NextIII \wedge \neg H(c,u)!AddAndSendOutgoingHTLC \wedge \neg H(c,u)!ReceiveUpdateAddHTLC$

   2.1. $\forall c \in C, u \in U : AC(c,u)!TimelockRegions' \subseteq AC(c,u)!TimelockRegions$

   Proof: Because there is no state that removes an HTLC, it holds that if a value is in the set $AC(c,u)!TimelockRegions$ then it is also in the set $AC(c,u)!TimelockRegions'$.

   2.2. $\forall c \in C, u \in U : H(c,u)!TimelockRegions' \subseteq H(c,u)!TimelockRegions$

   2.2.1. $\forall c \in C, u \in U : H(c,u)!T_1' \subseteq H(c,u)!T_1$

   Proof: Because the only action that adds a payment to *NewPayments* is *H(c,u)!ReceiveUpdateAddHTLC* and, by 2, it holds that $\neg H(c,u)!ReceiveUpdateAddHTLC$.

   2.2.2. $\forall c \in C, u \in U : H(c,u)!T_2' \subseteq H(c,u)!T_2$

   Proof: Because the only action that creates an `update_add_htlc` message is *H(c,u)!AddAndSendOutgoingHTLC* and, by 2, it holds that $\neg H(c,u)!AddAndSendOutgoingHTLC$.

   2.2.3. $\forall c \in C, u \in U : H(c,u)!T_3' \subseteq H(c,u)!T_3$

   Proof: Because an HTLC that has not been fulfilled in state t has already not been fulfilled in state s.

   2.2.4. Q.E.D.

   Proof: By definition of $H(c,u)!TimelockRegions$

   2.3. $\forall c \in C, u \in U : \{t + 1 \mid t \in AC(c,u)!TimeBounds\}' \subseteq relZTP^{LedgerTime}$

   Proof:

   2.3.1. $\forall c \in C, u \in U : \{t + 1 \mid t \in AC(c,u)!B_1\}' \subseteq relZTP^{LedgerTime}$

   Proof: Because $\{t + 1 \mid t \in AC(c,u)!B_1\}'$ is always a subset of $AC(c,u)!TimelockRegions'$.

   2.3.2. $\forall c \in C, u \in U : \{t + 1 \mid t \in AC(c,u)!B_2\}' \subseteq relZTP^{LedgerTime}$

Proof: There are only two different values that the set $AC(c,u)!B_2$ can assume: Before both users of the channel c are in the protocol state `closed`, the set has the value $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ and when the users are in the protocol state `closed`, the set has the value $\{MAX\_TIME\}$. Because the protocol state `closed` is the final protocol state that is not changed anymore, the value of the set $AC(c,u)!B_2$ changes only from $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ to $\{MAX\_TIME\}$.

2.3.3. Q.E.D.

Proof: By definition of $AC(c,u)!TimeBounds$.

2.4. Q.E.D.

Proof: By definition of $relZTP^{LedgerTime}$

3. Case: $H(c,u)!AddAndSendOutgoingHTLC \vee H(c,u)!ReceiveUpdateAddHTLC$

Proof: These actions add a an outgoing resp. incoming HTLC with state new. Further, $H(c,u)!AddAndSendOutgoingHTLC$ sends an `update_add_htlc` message and $H(c,u)!ReceiveUpdateAddHTLC$ adds a new payment to *NewPayments*.

3.1. $\forall c \in C, u \in U : AC(c,u)!TimelockRegions' \subseteq relZTP^{LedgerTime}$

Proof: Because the state of an HTLC does not change in the step, we only have to consider that $AC(c,u)!TimelockRegions'$ contains the timelock and the HTLC's timelock +*FORWARD_DELTA* + 1 of an HTLC in state new. If $H(c,u)!AddAndSendOutgoingHTLC$ then this timelock is the timelock of a payment in $NewPayments$ and, by definition of $H(c,u)!T_1$, the HTLC's timelock and the HTLC's timelock +*FORWARD_DELTA* + 1 are in the set $relZTP^{LedgerTime}$. If $H(c,u)!ReceiveUpdateAddHTLC$ this timelock is the timelock of an HTLC in an `update_add_htlc` message and, by definition of $H(c,u)!T_2$, the HTLC's timelock and the HTLC's timelock +*FORWARD_DELTA* + 1 are in the set $relZTP^{LedgerTime}$.

3.2. $\forall c \in C, u \in U : H(c,u)!TimelockRegions' \subseteq H(c,u)!TimelockRegions$

3.2.1. $\forall c \in C, u \in U : H(c,u)!T_1' \subseteq H(c,u)!T_1 \cup H(c,u)!T_2$

Proof: The only action that adds a payment to *NewPayments* is *H(c,u)!ReceiveUpdateAddHTLC*. Thus, if $H(c,u)!AddAndSendOutgoingHTLC$ it holds that $H(c,u)!T_1' = H(c,u)!T_1$. If $H(c,u)!ReceiveUpdateAddHTLC$, it holds that $H(c,u)!T_1' \subseteq H(c,u)!T_2$ because the absolute timelock of the payment is copied from the `update_add_htlc` message that is received.

3.2.2. $\forall c \in C, u \in U : H(c,u)!T_2' \subseteq H(c,u)!T_2 \cup H(c,u)!T_1$

Proof: The only action that creates an `update_add_htlc` message is *H(c,u)!AddAndSendOutgoingHTLC*. Thus, if $H(c,u)!ReceiveUpdateAddHTLC$, it holds that $H(c,u)!T_2' = H(c,u)!T_2$. If $H(c,u)!AddAndSendOutgoingHTLC$, it holds that $H(c,u)!T_2' \subseteq H(c,u)!T_1$ because, by definition of $H(c,u)!T_1$, all values that are calculated for a `update_add_htlc` message are contained in the set $H(c,u)!T_1$.

3.2.3. $\forall c \in C, u \in U : H(c,u)!T_3' \subseteq H(c,u)!TimelockRegions$

Proof: Because the state of an HTLC does not change in the step, we only have to consider that $H(c,u)!T_3'$ contains the timelock of an HTLC in state new. If

$H(c,u)!AddAndSendOutgoingHTLC$ then this timelock is the timelock of a payment in $NewPayments$ and, by definition of $H(c,u)!T_1$, the HTLC's timelock is in the set $H(c,u)!TimelockRegions$. If $H(c,u)!ReceiveUpdateAddHTLC$ this timelock is the timelock of an HTLC in an `update_add_htlc` message and, by definition of $H(c,u)!T_2$, the HTLC's timelock is in the set $H(c,u)!TimelockRegions$.

3.2.4. Q.E.D.

Proof: By definition of $H(c,u)!TimelockRegions$

3.3. $\forall c \in C, u \in U : \{t+1 \mid t \in AC(c,u)!TimeBounds\}' \subseteq relZTP^{LedgerTime}$

Proof:

3.3.1. $\forall c \in C, u \in U : \{t+1 \mid t \in AC(c,u)!B_1\}' = relZTP^{LedgerTime}$

Proof: Because $\{t+1 \mid t \in AC(c,u)!B_1\}'$ is always a subset of $AC(c,u)!TimelockRegions'$.

3.3.2. $\forall c \in C, u \in U : \{t+1 \mid t \in AC(c,u)!B_2\}' \subseteq relZTP^{LedgerTime}$

Proof: There are only two different values that the set $AC(c,u)!B_2$ can assume: Before both users of the channel c are in the protocol state `closed`, the set has the value $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ and when the users are in the protocol state `closed`, the set has the value $\{MAX\_TIME\}$. Because the protocol state `closed` is the final protocol state that is not changed anymore, the value of the set $AC(c,u)!B_2$ changes only from $\{MAX\_TIME - TO\_SELF\_DELAY, MAX\_TIME\}$ to $\{MAX\_TIME\}$.

3.3.3. Q.E.D.

Proof: By definition of $AC(c,u)!TimeBounds$.

3.4. Q.E.D.

Proof: By definition of $relZTP^{LedgerTime}$

4. Q.E.D.

By 2 and 3.

A.3.17. Proof of Lemma 14

We recall Lemma 14 which was defined on page 160:

Lemma 14. *For each channel $c \in C$, the mapping m_c^{II} maps each behavior $\sigma = \langle \sigma_0, \sigma_1, \sigma_2, ... \rangle$ of SpecificationII^{time} to a behavior $\langle m_c(\sigma_0), m_c(\sigma_1), m_c(\sigma_2), ... \rangle$ of SpecificationII$^{time,\ single}$ of channel c.*

Before we prove Lemma 14, we introduce Lemma 23 that we need in the proof of Lemma 14.

Lemma 23.

$$SpecificationII^{time,\ single} \Rightarrow \forall x \in \mathcal{X} : relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$$

Proof: Because *SpecificationII*$^{time,\ single}$ corresponds largely to *SpecificationII*^{time}, Lemma 23 can be proven analogously to the proof of Lemma 13 in Section A.3.7. The proof is analogous to the proof of Lemma 21 because *SpecificationII*$^{time,\ single}$ is defined based on *SpecificationII*^{time} analogously to how *SpecificationI*$^{time,\ single}$ is defined based on *SpecificationI*^{time}. Therefore, we do not repeat the proof here but refer to Section A.3.11 where Lemma 21 is proven.

We prove Lemma 14:

1. For all initial states of *SpecificationII*^{time} and all channels $c \in C$ it holds that $m_c^{II}(s)$ is an initial state of *SpecificationII*$^{time,\ single}$

   By definition of the initial states, applying the selection by m_c^{II} to an initial state of *SpecificationII*^{time} results in an initial state of *SpecificationII*$^{time,\ single}$.
2. Assume: s is a state of *SpecificationII*^{time} and $m_c^{II}(s)$ a state of *SpecificationII*^{time} and $s \rightarrow t$ is a step of *SpecificationII*^{time}

   Prove: $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$

   2.1. A step of *SpecificationII*^{time} is either (1) a step of *AbstractEnforcement* or *HTLCUser* for channel c, (2) a step of *AbstractEnforcement* or *HTLCUser* for a channel $c' \neq c$, (3) a step of *LedgerTime*, or (4) a final withdraw step.

   Proof: By definition of *NextIV*.

   2.2. Case: $s \rightarrow t$ is a step of *AbstractEnforcement* or *HTLCUser* for channel c

   To show that $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$, we distinguish between the channel-specific variables, user-specific variables, and global variables.

   2.2.1. The update of the channel-specific variables for channel c in step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$.

   The channel-specific variables are unchanged by the mapping m_c^{II}. The statement follows because the step $s \rightarrow t$ is a step of the modules *AbstractEnforcement* or *HTLCUser* and the same modules are used in *SpecificationII*$^{time,\ single}$.

   2.2.2. The update of the user-specific variables for user u in step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$.

   While some user-specific variables are unchanged by the mapping m_c^{II}, other variables (*UserPreimageInventory[u]*, *UserLatePreimages[u]*, *UserExtBalance[u]*, *UserPaymentSecretForPreimage[u]*, *UserPayments[u]*, *UserNewPayments[u]*, and *UserFailedPayments[u]*) are changed.
   Actions of the modules *AbstractEnforcement* and *HTLCUser* update the state of payments in the variable *UserPayments[u]*. All steps of these actions are also possible if the variable *UserPayments[u]* contains fewer entries.
   From *UserPreimageInventory[u]*, *UserLatePreimages[u]*, *UserPaymentSecretForPreimage[u]*, *UserPayments[u]*, *UserNewPayments[u]*, and *UserFailedPayments[u]* only entries are relevant that belong to a payment over channel c. The mapping m_c^{II} is defined to keep these values in the set and remove only values that do not belong to a payment that is routed over channel c.
   The variable *UserExtBalance[u]* is set by the mapping m_c^{II} to a user's funding amount before the channel is open, to 0 while a channel is open, and, after the

channel has been closed, to a user's on-chain balance that is related to channel c. This corresponds to the changes expected by the action of *AbstractEnforcement* to open a payment channel and by the final withdraw action.

2.2.3. The update of the global variables in step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$.

Two global variables are used by the modules *AbstractEnforcement* and *HTLCUser* and both variables are changed by the mapping m_c^{II}: *Messages* and *LedgerTime*.
The variable *Messages* is reduced to contain only messages that are sent or received by a user in the channel c. Thus, all messages that are relevant for an action of *SpecificationII*$^{time,\ single}$ are still contained in the variable after applying the mapping m_c^{II}.
For steps of the modules *AbstractEnforcement* and *HTLCUser* in *SpecificationII*$^{time,\ single}$, it must hold that the variable *LedgerTime* is unchanged. The value of $m_c^{II}(t).LedgerTime$ is defined as the maximum of $m_c^{II}(s).LedgerTime$ and all values in $relZTP^{LedgerTime}$ of *SpecificationII*$^{time,\ single}$ in state t that are lower than or equal to $t.LedgerTime$. We must show that the value of the variable *LedgerTime* is unchanged, i.e. that $m_c^{II}(s).LedgerTime = m_c^{II}(t).LedgerTime$. This holds if the largest value of $relZTP^{LedgerTime}$ of *SpecificationII*$^{time,\ single}$ in state t that is lower than or equal to $t.LedgerTime$ is lower than or equal to the largest value of $relZTP^{LedgerTime}$ of *SpecificationII*$^{time,\ single}$ in state s that is lower than or equal to $s.LedgerTime$. This holds if it holds in *SpecificationII*$^{time,\ single}$ that $relZTP^{LedgerTime'} \subseteq relZTP^{LedgerTime}$ which is shown by Lemma 23.

2.2.4. Q.E.D.

Because the steps 2.2.1, 2.2.2, 2.2.3 cover all variables of *SpecificationII*$^{time,\ single}$.

2.3. Case: $s \rightarrow t$ is a step of *AbstractEnforcement* or *HTLCUser* for channel $c' \neq c$

The variables in *SpecificationII*^{time} that are specific to channel c do not change between states s and t. However, the variables for the users participating in channel c and channel c' might change in the step $s \rightarrow t$ or global variables might change. For example, a user of channel c and channel c' might receive a preimage. To account for such changes, *SpecificationII*$^{time,\ single}$ contains the module *ChannelEnvironment*. The module *ChannelEnvironment* abstracts all actions that can happen in other payment channels.
The global variable *LedgerTime* cannot be changed in step $s \rightarrow t$ because $s \rightarrow t$ is a step of an action of the modules *PaymentChannelUser* or *HTLCUser*. Because, by Lemma 23, the set $relZTP^{LedgerTime}$ in *SpecificationII*$^{time,\ single}$ can only become smaller during the step $m_{c'}(s) \rightarrow m_{c'}(t)$, the value of *LedgerTime* in state $m_{c'}(t)$ must equal $m_{c'}(s).LedgerTime$, i.e. the value of *LedgerTime* does not change in the step $m_{c'}(s) \rightarrow m_{c'}(t)$.
The remaining global variables and user variables are changed by the modules *AbstractEnforcement* and *HTLCUser*. We have shown in Section A.3.10 that the changes of the modules *PaymentChannelUser* and *HTLCUser* to variables shared with other channels are described by the module *ChannelEnvironment*. Because the module *AbstractEnforcement* is an abstraction of the module *PaymentChannelUser*, it follows that

the module *ChannelEnvironment* also describes how the modules *AbstractEnforcement* and *HTLCUser* change variables shared between multiple channels.

2.4. Case: $s \rightarrow t$ is a step of the module *LedgerTime*

The only change between states s and t is that the value of the clock *LedgerTime* increases. By the definition of m_c^{II}, the value of *LedgerTime* can only increase in step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ and *LedgerTime* in state $m_c^{II}(t)$ is set to a value that is in $relZTP^{LedgerTime}$. Because the time bounds in *SpecificationII*$^{time,\ single}$ are, by definition, a subset of the time bounds in *SpecificationII*^{time}, the step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ complies with the time bounds of *SpecificationII*$^{time,\ single}$. Because all other variables are unchanged in the step $s \rightarrow t$ and the mapping m_c^{II} does not depend on the value of *LedgerTime*, these variables are also unchanged in the step $m_c^{II}(s) \rightarrow m_c^{II}(t)$. It follows that the step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is a step of *SpecificationII*$^{time,\ single}$.

2.5. Case: $s \rightarrow t$ is a final withdraw step of action *WithdrawBalancesAfterChannelClosed*

The final withdraw action *WithdrawBalancesAfterChannelClosed* is defined in *SpecificationII*$^{time,\ single}$ as in *SpecificationII*^{time}. The action changes the global variable *LedgerTime* to the constant *TERMINATION* to indicate that the system has terminated. This value is not changed by the mapping m_c^{II} and, thus, the corresponding condition is fulfilled for the step $m_c^{II}(s) \rightarrow m_c^{II}(t)$.
The variable *UserExtBalance[u]* is the only user variable that is updated in the step $s \rightarrow t$ and changed by the mapping m_c^{II}. In the step $s \rightarrow t$, the channel balance of all honest users is withdrawn from the channels and added to the corresponding variable *UserExtBalance[u]*. In *SpecificationII*$^{time,\ single}$, only a single channel is modeled and, thus, only a user's balance in the modeled channel can be added to the user's external balance while, in *SpecificationII*^{time}, the sum of the user's balances in multiple channels might be added to the user's external balance. To account for this difference, the variable *UserExtBalance[u]* is modeled as a mapping of balances for each channel. In *SpecificationII*$^{time,\ single}$, only the value in *UserExtBalance[u]* for the channel c is updated and, thus, the update of a user's external balance in the step $m_c^{II}(s) \rightarrow m_c^{II}(t)$ is allowed by the action *WithdrawBalancesAfterChannelClosed* of *SpecificationII*$^{time,\ single}$.

2.6. Q.E.D.

By 2.1, the cases above (2.2, 2.3, 2.4, and 2.4) cover all possible cases.

3. Q.E.D.

By induction using 1 and 2.

A.3.18. Proof of Lemma 16

We recall Lemma 16 which was defined on page 160:

Lemma 16. *A step of ChannelAbstraction in SpecificationIII^{single} is also possible if the variables that are reduced by m_c^{II} and are used in SpecificationIII contain additional values that are filtered out by m_c^{II}.*

We show that this property holds for every action of the module *ChannelAbstraction* and every variable that is reduced by m_c^{II}. As the mapping m_c^{II} is defined analogously to the mapping m_c^{II} between specifications *SpecificationI^{time}* and *SpecificationI$^{time,\ single}$*, parts of the proof of Lemma 10 in Section A.3.12 are similar to this proof.
The actions *RequestInvoice* and *ReceivePaymentHash* update the state of a payment in *UserNewPayments[u]*. Analogously, the action *ReceiverProcessesPayments* and the action *ExecutePayments* update a payment in *UserPayments[u]*. For these actions, it is irrelevant whether there are further payments in the set.
The actions *SendAddPaymentMessage* and *AddForwardedPayment* add or remove a payment from the set in the variable *UserNewPayments[u]* which is also possible if there are additional values in the set. Analogously, the action *GenerateAndSendPaymentHash* adds a preimage to the variable *UserPreimageInventory[u]* and the action *FailPayment* adds a payment to the variable *UserFailedPayments[u]*.
For the invariant defined in the module *ChannelAbstractionWithPreimages* that every step of the module *SpecificationIII* must maintain, it is relevant whether the hash or the id of a payment that is contained in the set *ChannelForwardedPayments[c]* is contained in the variable *UserPreimageInventory[u]*, *UserLatePreimages[u]*, or *UserFailedPayments[u]*. By definition these variables contain the hashes resp. ids of all payments that are forwarded over the channel c. Because the set *ChannelForwardedPayments[c]* can only contain payments that are forwarded over the channel c, a step of *SpecificationVa* is also possible if the sets *UserPreimageInventory[u]*, *UserLatePreimages[u]*, or *UserFailedPayments[u]* contain additional values because these additional values must be for payments that are not forwarded over the channel c.

A.3.19. Proof of Theorem 5

We recall Theorem 5 which was defined on page 161:

Theorem 5. *The function g^{II} is a refinement mapping from SpecificationII^{time} to SpecificationIII.*

1. For all s in the state space of *SpecificationII^{time}* it holds that the externally visible variables of state $g(s)$) equal those of state s

   Proof: The externally visible variables are for all users $u \in U$: *UserPayments[u]*, *UserExtBalance[u]*, *UserChannelBalances[u]*, and *UserHonest[u]*. These variables are by definition of g unchanged by g.
2. For all initial states s of *SpecificationII^{time}*, $g(s)$ is an initial state of *SpecificationIII*

   Proof: All variables in state s are mapped by g either by directly assigning the value of the variable in state s or by applying the refinement mapping f. By definition of f, f does not change values in the initial states. By definition of *Init* of *SpecificationII^{time}* and

SpecificationIII, it follows that all variables of a state of *SpecificationII*^{time} are mapped to values of an initial state of *SpecificationIII*.

3. For all steps $s \rightarrow t$ of *SpecificationII*^{time}, $g(s) \rightarrow g(t)$ is a step of *SpecificationIII*.

   Proof:

   A step of *SpecificationII*^{time} can be one of the following.

   3.1. Case: $s \rightarrow t$ is a step of an action of the module *LedgerTime* in *SpecificationII*^{time}.

   Prove: $g(s) \rightarrow g(t)$ is a step of the module *LedgerTime* in *SpecificationII*.

   Proof: This proof is analogous to the corresponding proof step of the proof of Theorem 3 in Section A.3.13.

   3.2. Case: $s \rightarrow t$ is a step of an action of the modules *HTLCUser* or *AbstractEnforcement* in *SpecificationII*^{time}.

   Prove: $g(s) \rightarrow g(t)$ is a step of *SpecificationIII*.

   Proof: By Lemma 14, the step $s \rightarrow t$ can be mapped to a step in an instance of *SpecificationII*$^{time,\ single}$ for each channel c. By Lemma 15, these mapped steps for each channel c are mapped to steps of *SpecificationVa*. By Lemma 16, each step of a channel in *SpecificationVa* is also possible when the channel is composed with other channels in *SpecificationIII*. Because each step of the modules *HTLCUser* or *AbstractEnforcement* is a step only in one channel and a stuttering step in all other channels, it follows that the step $g(s) \rightarrow g(t)$ is a step of *SpecificationIII*.

   3.3. Case: $s \rightarrow t$ is a step of the action *WithdrawBalancesAfterChannelClosed* in *SpecificationII*^{time}.

   Prove: $g(s) \rightarrow g(t)$ is a step of *SpecificationIII*.

   The action *WithdrawBalancesAfterChannelClosed* in specifications *SpecificationII*^{time} and *SpecificationIII* are defined as compositions of the corresponding steps in the specifications *SpecificationII*$^{time,\ single}$ and *SpecificationVa*. For the channel specific variables, this can be directly seen. The only non-channel specific variable is the variable *UserExtBalance[u]*. This new value of an honest user's external balance is defined in each single channel specification (*SpecificationII*$^{time,\ single}$ and *SpecificationVa*) as the user's external balance for the channel. In the multi-channel specifications (*SpecificationII*^{time} and *SpecificationIII*), a user's external balance is defined as the sum of a user's external balances for all channels. Thus, it follows based on Lemma 15 that the step $g(s) \rightarrow g(t)$ is a step of *SpecificationIII*.

   3.4. Q.E.D.

   Proof: By 3.1, 3.2, 3.3 and because these cases cover all possible steps in *SpecificationII*^{time}.

4. Q.E.D.

   Proof: By 1, 2, 3, and the definition of refinement mappings and because *SpecificationIII* contains the same fairness condition as *SpecificationII*^{time} that only behaviors are valid that end in a state in which all users have withdrawn their balance.

A.3.20. Proof of Lemma 17

We recall Lemma 17 which was defined on page 163:

Lemma 17. *In SpecificationIII it holds that:*

$$\forall x, y \in \mathcal{X}, d \in \mathbb{N}_0, s \in \hat{\Sigma} : B^x(s) = B^x(T_d^y(s))$$

Proof: $B^{LedgerTime}$ is defined by *TimeBounds* of *ChannelAbstraction*. As the variable *LedgerTime* does not occur in the definition of $TimeBounds$, the definition of $TimeBounds$ is independent of *LedgerTime* and, thus, Assumption 4.1 holds for $B^{LedgerTime}$.

A.3.21. Proof of Lemma 18

We recall Lemma 18 which was defined on page 163:

Lemma 18. $relZTP^x$ *of SpecificationIII*^{time} *meets Assumption 4.2, i.e.* $\forall x \in \mathcal{X}, s \in \hat{\Sigma} : ZTP^x(s) \subseteq relZTP^x(s)$.

The set $relZTP^{LedgerTime}$ is defined as $relZTP$ in *SpecificationIII*^{time} which is statically defined as the timelocks and the timelocks plus $FORWARD\_DELTA$ for all channels on all initially defined payments' routes.

In the proof, we use Definition 23 as introduced in the proof in Section 4.3.1.

Proof: The module *ChannelAbstraction* contains two actions that depend on *LedgerTime*:

1. $ZTP^{LedgerTime}_{SendAddPaymentMessage}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *SendAddPaymentMessage* adds the payment with the lowest timelock of all payments whose timelock is lower than *LedgerTime*. Thus, the set $ZTP^{LedgerTime}_{SendAddPaymentMessage}(s)$ contains the timelock of all payments that can be added to the channel. By definition of $relZTP^{LedgerTime}$, these points in time are contained in the set $relZTP^{LedgerTime}$.
2. $ZTP^{LedgerTime}_{ExecutePayments}(s) \subseteq relZTP^{LedgerTime}(s)$
   Proof: The action *ExecutePayments* depends on the value of *LedgerTime* to choose the new state of payment and to choose whether a possibly received preimage is added to *UserLatePreimages[u]*. A preimage is added to *UserLatePreimages[u]* if the value of *LedgerTime* is greater than or equal to the payment's timelock plus $FORWARD\_DELTA$. Thus, $ZTP^{LedgerTime}$, contains each payment's timelock plus $FORWARD\_DELTA$. This value is included in the set $relZTP^{LedgerTime}$. A payment may be aborted if *LedgerTime* is greater than or equal to the payment's timelock and a payment must be aborted if *LedgerTime* is greater than or equal to the payment's timelock plus $FORWARD\_DELTA$. By definition of $relZTP^{LedgerTime}$, these values are contained in the set $relZTP^{LedgerTime}$.
3. For each action A of the module *LedgerTime* it holds that $ZTP^{LedgerTime}_{A}(s) \subseteq relZTP^{LedgerTime}(s)$

Proof: By the proof on page 326.

4. Q.E.D.

   By the steps 1, 2, and 3 and we have proven for all subactions of $InnerNext$ that $\forall x \in \mathcal{X}, s \in \Sigma : ZTP^x(s) \subseteq relZTP^x(s)$.

A.3.22.Proof of Lemma 19

We recall Lemma 19 which was defined on page 163:

Lemma 19. *For any two states s, t of SpecificationIII and all clocks $x \in \mathcal{X}$ (i.e., the clock LedgerTime), it holds that*

$$\textit{NextIII} \Rightarrow relZTP^{x\prime} \subseteq relZTP^x$$

Proof: Because $\textit{SpecificationIII}^{time}$ defines $relZTP$ independently of the current state and the definition only depends on constants.