

Blockchain Scalability through Secure Optimistic Protocols

Eckey, Lisa
(2020)

DOI (TUprints): <https://doi.org/10.25534/tuprints-00014041>

License:



CC-BY 4.0 International - Creative Commons, Attribution

Publication type: Ph.D. Thesis

Division: 20 Department of Computer Science

Original source: <https://tuprints.ulb.tu-darmstadt.de/14041>



TECHNISCHE
UNIVERSITÄT
DARMSTADT

BLOCKCHAIN SCALABILITY THROUGH SECURE OPTIMISTIC PROTOCOLS

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

von

Lisa Eckey, M.Sc.
geboren in Hamm (Westf.)

Gutachter: Prof. Dr. Sebastian Faust
Prof. Dr. Matteo Maffei

Datum der Einreichung: 27.02.2020

Datum der mündlichen Prüfung: 24.04.2020

Author: Lisa Eckey
Title: Blockchain Scalability through Secure Optimistic Protocols
Ort: Darmstadt, Technische Universität Darmstadt

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
<http://tuprints.ulb.tu-darmstadt.de>
tuprints@ulb.tu-darmstadt.de


Bitte zitieren Sie dieses Dokument als:
URN: [urn:nbn:de:tuda-tuprints-140412](https://nbn-resolving.org/urn:nbn:de:tuda-tuprints-140412)
URI: <https://tuprints.ulb.tu-darmstadt.de/id/eprint/14041>

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:
Attribution 4.0 International (CC BY 4.0)
<https://creativecommons.org/licenses/by/4.0/>



Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.



Wissenschaftlicher Werdegang

Oktober 2009 bis September 2012 Studium der Wirtschaftsinformatik (Bachelor of Science) an der Westfälischen-Wilhelms Universität Münster

April 2013 bis April 2016 Studium der IT-Sicherheit/ Netze und Systeme (Master of Science) an der Ruhr-Universität Bochum

April 2016 bis Oktober 2017 Beginn des Promotionsstudiums an der Ruhr-Universität Bochum in der Arbeitsgruppe für angewandte Kryptografie bei Prof. Sebastian Faust

November 2017 bis April 2020 Fortsetzung des Promotionsstudiums an der Technischen Universität Darmstadt am Lehrstuhl für angewandte Kryptografie bei Prof. Sebastian Faust

List of Publications

- [30] R. Böhme, L. Eckey, N. Narula, T. Moore, T. Ruffing, and A. Zohar. “Responsible Vulnerability Disclosure in Cryptocurrencies”. In: *Communications of the ACM (CACM)*. 10. 2020, pp. 62–71.
- [59] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi. “FastKitten: Practical Smart Contracts on Bitcoin”. In: *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA, USA, 2019, pp. 801–818.
- [67] S. Dziembowski, L. Eckey, and S. Faust. “FairSwap: How To Fairly Exchange Digital Goods”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada, 2018, pp. 967–984.
- [68] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. “Multi-party Virtual State Channels”. In: *Proceedings of the 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Darmstadt, Germany, 2019, pp. 625–656.
- [69] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. “Perun: Virtual Payment Hubs Over Cryptocurrencies”. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 2019, pp. 327–344.
- [72] L. Eckey, S. Faust, and J. Loss. “Efficient Algorithms for Broadcast and Consensus Based on Proofs of Work”. In: *IACR Cryptology ePrint Archive (2017)*. URL: <https://eprint.iacr.org/2017/915>.
- [73] L. Eckey, S. Faust, and B. Schlosser. “OptiSwap: Fast Optimistic Fair Exchange”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. Taipei, Taiwan, 2020, pp. 543–557.

Acknowledgments

The four years of being a doctoral student was a true journey for me – starting in Bochum and ending in Darmstadt, it brought me to many exciting destinations on the way. I am grateful to everyone who accompanied me on this path, guided, or enjoyed it with me. First of all, I thank my supervisor Sebastian Faust, who encouraged me to pursue this path. He inspired my interest in cryptography research, and I highly appreciated his guidance and feedback. I also thank him and all my co-authors for sharing and discussing research ideas with me and many successful collaborations. I enjoyed being a part of the open and welcoming cryptography community, where I met so many inspiring researchers and practitioners.

I am grateful to my disputation committee and especially to Matteo Maffei, who co-refereed my thesis for the helpful comments and feedback. For the financial support, I thank the DFG as well as the Universities in Bochum and Darmstadt, which provided the necessary infrastructure and resources for my research. I am especially thankful for the great opportunities to travel to summer schools and conferences, which allowed me to broaden my personal and academic horizons. I am very grateful to Jacqueline Wacker for her pleasant and enormously helpful support in navigating the bureaucratic university processes.

I enjoyed all the great moments with my colleagues and friends that accompanied me during this journey. Thank you for making my time in Bochum and Darmstadt enjoyable through stimulating coffee break discussions, horrible movies, joint trips, and fun events. I cannot say how much your virtual encouragement video cheered me up right before my defense. In particular, I thank my dear colleagues, travel companions, and friends Clara Paglialonga and Kristina Hostáková. I am glad that the three of us experienced this together, and I will never forget the memories we created in and outside of work. Without your cheerful encouragement, I would not have overcome the exhausting and disappointing moments of the past years.

My deep gratitude goes to my friends and family that encouraged me on this path. I especially thank Mario and Hannah for their reassuring trust in my abilities, Stefan for his patience and moral support, and of course, my parents, brother, and sister for their unconditional love and support.

Contribution

The results of this thesis are the outcome of collaborations and joint works with my supervisor Sebastian Faust and my co-authors, whom I would like to thank for inspiring discussions and contributions.

The results from Chapter 4 are based on the publication “Perun: Virtual Payment Hubs Over Cryptocurrencies” [69], which was developed in collaboration with Stefan Dziembowski, Sebastian Faust, and Daniel Malinowski. The paper was rewritten many times to improve the comprehensibility and to simplify the model, which makes it hard to distinguish the independent contributions. The version that appears in this thesis has minor modifications in the protocol description and explanations and a different structure, but otherwise contains verbatim passages from the published version [69] presented at the 40th IEEE Symposium on Security and Privacy (S&P) 2019. The high-level descriptions and the full security proof do not appear elsewhere and are exclusively my contribution. The implementation was iteratively done by Daniel Malinowski and me.

The protocol, which is described in Chapter 5, also developed through intense and inspiring discussions with Sebastian Faust and Stefan Dziembowski, is based on the paper “FairSwap: How To Fairly Exchange Digital Goods” [67]. The specification and write-up of this paper was primarily my contribution. In particular, this includes the modeling, the formal protocol description, the security proof, and implementation. Therefore, this thesis contains verbatim text from the original work [67] published in the proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security 2018.

The results of Chapter 6 are based on the publication “FastKitten: Practical Smart Contracts on Bitcoin” [59] (presented at the Usenix Security Symposium 2019) which is joint work with Poulami Das, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. My contribution to this paper was the overall idea and the protocol design. Leading inputs of my co-authors can be summarized as follows: Kristina Hostáková played a leading role in contributing to the security definitions and proof. The implementation and benchmarking were provided by Tommaso Frassetto, who

is also the leading author of the implementation section. Patrick Jauernig contributed the device side architecture, including the setup of a trusted execution enclave. To account for the split contribution, this thesis does contain verbatim text passages from the original paper except for the security and implementation sections (cf. Sections 7 and 8 of [59]).

Abstract

The digital currency Bitcoin has become a popular payment technology since its invention in 2008. Countless other projects have adopted and expanded the functionality of the underlying blockchain technology. These so-called cryptographic currencies allow users to send financial transactions over a decentralized global network. Some of these currencies even support payments that are based on complex conditions, also called smart contracts. The biggest obstacle to the practical use of cryptographic currencies is their limited scalability. Without a solution to this problem, blockchain technology cannot support the continuously growing user base or compete with centralized payment providers. This thesis presents three approaches to scaling that increase the number of transactions or enable a cheaper and faster execution of smart contracts.

The first contribution of this thesis is the PERUN protocol, which allows a network of users to send a large number of microtransactions at no cost. For this purpose, all users of the system open a so-called payment channel once and use it to send off-chain transactions without costs or delays. We will also show how to combine these channels in an off-chain manner to so-called virtual channels that connect even more users. The next contribution of this dissertation is the FAIRSWAP protocol, which aims at reducing the costs for the secure sale of large digital goods. It improves the scalability of such “fair exchange” protocols by reducing both the storage requirements and the complexity of the underlying smart contracts. We then present another protocol called FASTKITTEN, which uses a Trusted Execution Environment (TEE) to secure the off-chain execution of smart contracts. A TEE provides a secure runtime environment in which programs are executed safely and correctly. This allows an operator to execute the smart contracts on inputs from the users off-chain, which makes the execution much faster and cheaper for all participants.

To guarantee the security of these protocols, each construction is accompanied by detailed formal security definitions and cryptographic proofs. Furthermore, we demonstrate the efficiency of the protocols by implementing and analyzing the costs of each protocol.

Zusammenfassung

Die digitale Wahrung Bitcoin hat sich seit ihrer Erfindung im Jahr 2008 zu einer popularen Zahlungstechnologie entwickelt. Das hohe Interesse an der zugrundeliegenden Blockchain-Technologie wird vor allem durch die zahlreichen Projekte verdeutlicht, die in den letzten zehn Jahren die Funktionalitat von Bitcoin ibernommen und erweitert haben. Diese so genannten kryptographischen Wahrungen ermoglichen es, den Benutzern, finanzielle Transaktionen iber ein globales, dezentralisiertes Netzwerk zu versenden. Einige dieser digitalen Wahrungen ermoglichen sogar Zahlungen, die an komplexe Bedingungen geknupft werden, welche durch sogenannte Smart Contracts beschrieben werden. Das grote Hindernis fur den praktischen Einsatz von kryptographischen Wahrungen ist ihre mangelnde Skalierbarkeit. Ohne eine Losung fur dieses Problem kann die Blockchain Technologie die standig steigenden Nutzerzahlen nicht unterstutzen und nicht mit zentralisierten Zahlungsanbietern konkurrieren. In dieser Arbeit werden drei Losungsansatze zur Skalierung vorgestellt, die es ermoglichen viele Transaktionen und komplexe Smart Contracts gunstiger und schneller zu abzuwickeln.

Der erste Beitrag dieser Arbeit ist das PERUN-Protokoll, das es einem Netzwerk von Nutzern erlaubt, eine groe Anzahl von Mikrotransaktionen kostenlos zu versenden. Zu diesem Zweck oeffnen alle Benutzer des Systems einmalig einen sogenannten Zahlungskanal und nutzen diesen, um Zahlungen zwischen den Nutzern direkt und ohne Kosten oder Verzogerungen auszufuehren. Das PERUN Protokoll ermoglicht es auerdem diese Kanale ohne Blockchain Interaktionen zu so genannten virtuellen Kanalen zu kombinieren, die noch mehr Nutzer verbinden.

Der nachste Beitrag dieser Dissertation ist das FAIRSWAP-Protokoll, das zum Ziel hat die Kosten fur den sicheren Verkauf von groen digitalen Gutern zu senken. Dabei wird die Skalierbarkeit solcher “Fair Exchange”-Protokolle verbessert indem sowohl der Speicherbedarf als auch die Komplexitat der zugrundeliegenden Smart Contracts reduziert wird.

Der dritte Beitrag dieser Dissertation ist ein Protokoll namens FASTKITEN, das Trusted Execution Environments (TEEs) verwendet, um die off-chain Ausfuehrung von Smart Contracts abzusichern. TEEs bieten eine abgesicherte Laufzeitumge-

bung in denen Programme sicher und korrekt ausgeführt werden. Sie erlauben es einem sogenannten Operator, die Smart Contracts auf der Grundlage von Eingaben der Benutzer lokal auszuführen und damit Kosten und Laufzeiten senkt.

Formale Sicherheitsdefinitionen und kryptographische Beweise garantieren die Sicherheit der entwickelten Protokolle. Des Weiteren zeigen wir die Effizienz der Protokolle indem wir eine Implementierung anfertigen und analysieren.

Contents

1	Introduction	1
1.1	Challenges for Blockchain Technology	2
1.2	Goal of this Thesis	5
1.3	Contribution	6
1.4	Structure of this Thesis	9
2	Cryptographic Preliminaries	11
2.1	Notation	11
2.2	Provable Security	12
2.3	Cryptographic Primitives	14
2.4	The Universal Composability Model	18
3	Blockchain Technology	23
3.1	Bitcoin	24
3.1.1	Transactions	24
3.1.2	Blocks and Mining	25
3.1.3	Bitcoin Fees	28
3.2	Ethereum and Smart Contracts	29
3.2.1	Smart Contracts	29
3.2.2	Designing Secure Contracts	32
3.3	Scalability Solutions	33
3.3.1	Changing Blockchain Parameters	34
3.3.2	Scaling the Consensus Layer	34
3.3.3	Scaling Through Off-Chain Protocols	36
3.4	Formal Treatment of the Blockchain	41
3.4.1	Security Provided by the Public Ledger	42
3.4.2	Communication Model	43
4	Virtual Payment Channel Hubs	45
4.1	Overview	47
4.1.1	Intuition and Design Ideas	48

Contents

4.1.2	Related Work	56
4.2	Preliminaries	59
4.2.1	Channels Syntax	60
4.2.2	The Ledger Functionality	61
4.3	Ideal Functionality	62
4.3.1	Restrictions to the Environment	65
4.3.2	Perun Properties	66
4.3.3	Formal Security Statement	68
4.4	The PERUN Protocol	68
4.4.1	The Channel Smart Contract \mathcal{C}	70
4.4.2	The Π_{channel} protocol	74
4.5	PERUN Security Proof	86
4.5.1	Ledger Channel Opening	89
4.5.2	Channel Updating	90
4.5.3	Ledger Channel Closing	91
4.5.4	Virtual Channel Opening	93
4.5.5	Virtual Channel Closing	94
4.6	Implementation and Performance	96
4.6.1	Execution Times	96
4.6.2	Implementation and Gas Costs	96
4.6.3	Channel Network Comparison	98
4.7	Discussion and Extension	98
4.7.1	Extensions and Impact	99
5	Moving Complex Computation Off-Chain	103
5.1	Overview	104
5.1.1	Intuition and Design Ideas	105
5.1.2	Related Work	111
5.2	Preliminaries	113
5.2.1	Modeling Circuits	113
5.2.2	The Ledger Functionality	114
5.2.3	Global Random Oracle	115
5.2.4	Constructing the FAIRSWAP Building Blocks in the Random Oracle Model	118
5.2.5	Commitment Scheme Construction	121
5.3	Ideal Functionality for Coin aided Fair Exchange	122

Contents

5.4	FAIRSWAP Protocol	124
5.4.1	Witness and Transcript Encoding Scheme	126
5.4.2	The Judge Smart Contract \mathcal{C}	130
5.4.3	The Witness Selling Protocol Π_{FAIRSWAP}	131
5.5	FAIRSWAP Security Proof	132
5.5.1	Informal Security Analysis	133
5.5.2	Formal GUC Security Proof	135
5.6	Implementation and Performance	149
5.7	Discussion and Extension	155
5.7.1	Countermeasure against Free-Riding	155
5.7.2	Interactive Dispute	156
5.7.3	Splitting Escrow and Judge Function	156
5.7.4	Setting Financial Incentives for Honest Behavior	158
5.7.5	FAIRSWAP in Channels	159
6	Off-Chain Smart Contracts on Bitcoin	161
6.1	Overview	161
6.1.1	Intuition and Design Ideas	163
6.1.2	Related Work	167
6.2	Preliminaries	171
6.2.1	Modeling the Blockchain.	172
6.2.2	Modeling the TEE.	173
6.3	Security Properties	174
6.4	The FASTKITTEN Protocol	176
6.4.1	Setup Phase	176
6.4.2	Round Computation Phase	180
6.4.3	Finalize Phase	181
6.5	Security Evaluation	182
6.6	Implementation and Performance	185
6.6.1	Implementation Challenges	185
6.6.2	Performance	186
6.7	Discussion and Extensions	188
6.7.1	Privacy	188
6.7.2	Applications	189
6.7.3	Fees for the Operator	190
6.7.4	Incentive-driven Adversary	191
6.7.5	Fault Tolerance	191

Contents

6.7.6 Multi-Currency Contracts	192
7 Conclusion	193
List of Figures	197
List of Tables	199
List of Abbreviations	201
Bibliography	203

1 Introduction

Blockchain technology emerged in 2008 when Satoshi Nakamoto proposed a cryptographic currency for trustless online payments called *Bitcoin* [151]. In contrast to previously proposed electronic cash protocols [49, 50, 32], Bitcoin does not require trust in a bank or otherwise centralized authority. Instead, the system is secured by a cryptographic protocol that is executed in an open peer-to-peer network. Motivated by the ongoing financial crisis, Bitcoin promised to be the “cash of the Internet”, which cannot be controlled by financial institutions.

The Bitcoin implementation was deployed in 2009, only a few months after Nakamoto published his protocol on a cryptography mailing list [150]. Initially, Bitcoin was mostly used by a handful of enthusiasts, but it gained considerable popularity over time when an increasing number of users adopted the currency¹. The technology also received widespread attention from both academia and industry which lead to many projects analyzing, applying, or extending Nakamoto’s ideas. The broad interest in this technology is illustrated by the high amount of projects and research papers in this area, e.g., the original Bitcoin paper was cited around 9000 times [26] and its implementation was forked over 25 thousand times [182].

Bitcoin and its follow-up projects are also called *cryptocurrencies* or *distributed ledger technologies*. Their foremost goal is to provide secure financial transactions in decentralized, trustless networks. The main building block of cryptocurrencies is the *blockchain*, a public ledger, which stores the history of all transactions. Nodes of the underlying peer-to-peer network, called *miners*, collect transactions and publish them in new blocks, which extend the chain. Once transactions appear in this immutable public log, they are considered valid. The basis for distributed ledger technologies form established cryptographic primitives, i.e., *digital signatures* authenticate the transfer of coins and *hash functions* link the blocks together. But the main challenge of the system is that it requires all miners to reach consensus in an open and unregulated network. These, *permissionless systems* are often vul-

¹Today, Bitcoin’s market capitalization exceeds 140 billion euros, which is roughly equal to the GDP of Bulgaria [154].

1 Introduction

nerable to so-called *Sybil attacks* [65], in which miners get an unfair advantage by creating new fake identities. Many cryptocurrencies protect against these attacks by letting miners solve a cryptographic puzzle, called a Proof of Work (PoW) [12]. PoWs ensure that the chance of proposing a new block is proportional to the computational resources a miner is willing to invest. The system's security relies on the assumption that honest parties control the majority of the computational power.

Smart Contracts. Blockchain technology is also the basis for other promising innovations, most notably *smart contracts*, which allow users to deploy programming code on the blockchain. Smart contracts can store data publicly, receive coins, and define rules on their redistribution. The blockchain enforces these rules, which allow developers to build self-enforcing trustless applications. While Bitcoin only has limited support for smart contracts, other cryptocurrencies, like Ethereum [186], have incorporated powerful contract programming languages in their design. Popular applications of smart contracts can be found in the sharing economy [18, 174], e-commerce [6], trading [126, 10], online gambling [82, 63] and digital rights management [66]. When designing protocols between mutually untrusted parties, smart contracts can be utilized in various ways to secure the correct and fair protocol execution.

1.1 Challenges for Blockchain Technology

While some have regarded Nakamoto's original protocol as innovative and groundbreaking, it also received quite some negative attention, and countless projects work on adding additional features. Bitcoin and other cryptocurrencies have been associated with crime because of their pseudonymous user identification. For example, Bitcoin was the dominant payment method for online black markets [54] and ransom payments of extortion malware [113]. While some research papers (e.g., [143]) showed how law enforcement could follow the paths of pseudonymous transactions and deanonymize payments, other projects [146, 181, 171] proposed new currencies with advanced privacy features, that promise anonymous and unlikable payments. Another criticism that cryptocurrencies face is their enormous energy consumption. As more miners joined the network and invested their computational power, the energy usage of the overall Bitcoin system increased drastically. Currently, the energy consumption and carbon footprint of Bitcoin equals that of a small country [64]. This waste of natural resources motivated a large body of

1 Introduction

works that propose more sustainable cryptocurrencies, i.e., based on alternative mining puzzles [116, 149].

Scalability. But one of the biggest disadvantages of distributed ledger technologies is their limited scalability, which also motivates this thesis. The original paper envisioned Bitcoin to provide “cheap and fast payments” [151] which, as the user base grew, quickly turned out to be unrealistic and can not be provided by the blockchain technology as it is in use today. Ethereum also suffers from limited scalability, which has been called a “big bottleneck”, that could hinder the system’s further adoption [129].

The limited scalability of cryptocurrencies results from two inherent factors: the *block creation time* and the *fixed block size*. The former factor influences the average time between the creation of two blocks, and the later limits their maximum size. Both limitations are necessary as they ensure that blocks have sufficient time to be distributed in the peer-to-peer network. Even with both security measures in place, temporary blockchain forks can occur when two miners propose different blocks (somewhat) simultaneously. The consensus rules, which say that miners should always extend the longest chain, ensure that miners will eventually agree on one of the chains. To protect against this temporary uncertainty, users should only accept new blocks and the transactions inside them, once a few newer blocks extend it; i.e., in Bitcoin, it can take around 60 minutes until transactions are confirmed. In addition to the long delays, blocks can only include transactions up to a maximal block size. This parameter either upper bounds the data size (as in Bitcoin) or the complexity of instructions (as in Ethereum). However, in both cases, there is a limit to the overall transaction throughput, which indicates the number of transactions supported per second. In Bitcoin, it is said to be around seven transactions per second [58] and 15 for Ethereum [17]. These numbers are incomparable to the high throughput of centralized systems like the Visa credit card network, which support thousands of transactions per second [58]. During times of high transaction volume, many transactions compete for the limited space in blocks. As a result, transaction fees increase drastically as only the payments with the most lucrative fees are chosen by the miners. The combination of long confirmation delays and unpredictable and high transaction fees make blockchain technology unattractive for many applications.

Many research papers have focused on these scalability issues and have proposed solutions on multiple layers. The authors of [58] have analyzed the limits of how far the blockchain parameters can be tweaked to ensure that blocks can still propagate

1 Introduction

fast enough in the underlying network. Some scaling proposals change the rules of the blockchain system, while others change the way that users interact with it. An example to a more scalable consensus proposal is *sharding* [133, 119, 189], which partitions the blockchain into multiple shards, that are maintained by a subset of nodes. As every miner only has to verify a small subset of transactions and blocks, this approach can scale the overall transaction throughput, if the honest majority assumption holds for each sub-committee of the network. Another idea is to arrange transactions in a *Directed Acyclic Graph (DAG)* instead of a blockchain [162], such that they directly reference (approve) previous transactions. This allows the network to be asynchronous and have (to some degree) different views on the currently approved transactions.

Another group of proposals build scaling improvements without changing the consensus rules and is often classified as second layer or *off-chain* research. The key idea of protocols proposed in this area is to reduce the on-chain transaction load by letting the parties interact directly with each other, instead of sending transactions to the blockchain. As a result transaction costs and confirmation times can be reduced. Examples of off-chain protocols are payment channels (e.g., Lightning [161]), commit chains (e.g., Plasma [160]), and off-chain execution frameworks (e.g., TrueBit [177]).

Secure Protocol Design. As these protocols can become rather complex, it is challenging to design them without flaws. Severe attacks like the 47 million euros theft from the “DAO” smart contract [9] or the 30 million euros theft from Parity wallets [156] show the importance of secure contract and protocol design. The field of modern cryptography provides important tools necessary to achieve high confidence about the security of a protocol. By formally defining the properties of a system, we can quantify its security and describe the attacks we aim to protect against. Then we can build a protocol and prove that it satisfies these requirements. By writing formal proofs, we analyze the security of the protocol and capture possible attack vectors during the design phase.

The need for schemes that are *secure by design* is demonstrated by various examples of flawed schemes, that have been designed in an ad-hoc fashion, without undergoing a thorough security analysis. A famous example for such a project is the initial proposal [98] of the Transport Layer Security (TLS) protocol and its predecessor, Secure Sockets Layer (SSL), which are essential cryptographic protocols securing communication over the Internet. It was later shown that the initial proposals included severe conceptual flaws and vulnerabilities [145]. In the

context of cryptocurrencies, the need for secure design has been highlighted by, e.g., the Zerocoin project [146], in which it was possible to destroy coins of honest users [170]. This flaw was not captured in the original proposal, despite the fact that it included a correct security proof, because its security definition did not capture such an attack.

1.2 Goal of this Thesis

The goal of this thesis is to increase blockchain scalability through the use of smart contracts. In order to ensure security, we apply the methods of modern cryptography and develop formal security definitions and proofs for our protocols. We utilize the existing blockchain protocols, in particular, Bitcoin and Ethereum, and build off-chain protocols that shift the main transaction load away from the blockchain. We apply a scaling technique from the area of Multi-Party Computation (MPC) that makes complex protocols more efficient. These so-called *optimistic protocols* [8] were initially proposed for fair exchange settings, where two parties want to exchange two values. Fair exchange ensures that either both parties learn the respective values or neither party learns the input of the other. It has been shown that it is impossible to build such protocols without an (often expensive) Trusted Third Party (TTP) [155]. Optimistic protocols distinguish two cases: When both parties behave honestly, we call the protocol execution *optimistic*, while the *pessimistic* case occurs when at least one player starts deviating from the honest behavior. The idea is to rely on the trusted intermediary only in the pessimistic case, and make the optimistic case efficient and cheap. The design rationale behind this setup is that the optimistic case is much more likely to occur, and thus the protocol execution will be efficient in the standard case. The trusted intermediary can act as a judge in many cases and identify who of the participants misbehaved. This party could then be punished, and the other party can be compensated. This idea is often applied when optimistic protocols are used in the context of blockchain technologies where financial penalties and compensation is easy [23]. In the protocols designed in this thesis, we aim to minimize interaction with the blockchain in the optimistic case and only rely on it to judge on misbehavior in the pessimistic case.

An additional challenge for our protocols is that the concept of monetary transactions, coins, or financial security is traditionally not considered by modern cryptography. As these elements are crucial to the secure design of our protocols, we model the blockchain and the interface that we require from it. To this end, we

formally define the ledger system, which takes care of coins on the blockchain and ensures their correct transfer. We also provide a formal definition of smart contracts, which includes their functionality and all interactions with both the ledger and the protocol participants. The protocols we design are accompanied by formal proofs that show how we achieve the defined security guarantees. In particular, we ensure that a party will not lose its coins if it behaves honestly, even if all other involved parties do not follow the defined protocol. This means that no malicious party can benefit from cheating.

1.3 Contribution

We design three protocols in this thesis that use optimistic protocols for scaling blockchain technologies. We will discuss them in more detail in Chapters 4 to 6. In all of these protocols we will try to minimize (expensive and slow) interactions with the blockchain to reduce the costs. We provide formal security analysis for each of them and show feasibility through a proof-of-concept implementation.

Chapter 4: Virtual Payment Channel Hubs. In Chapter 4, we present the PERUN protocol, based on the publication “Perun: Virtual payment hubs over cryptocurrencies” [69] published at the 2019 IEEE Symposium on Security. In this work, the scalability of blockchains is improved by extending the functionality of payment channels. Payment channels were previously introduced by [61, 161, 148] and allow two parties to send transactions off-chain to each other directly instead of sending them on the blockchain. In Ethereum, payment channels are secured by a smart contract that is only required for the setup and closing of channels in the optimistic case, which makes this case very cheap and efficient. The pessimistic case occurs whenever the two channel participants disagree on the state of the channel, or if one of the two parties aborts. In this case, a party can start a dispute process and complain to the smart contract. However, we assume that the pessimistic case will rarely happen as no party can benefit from it. Thus, channels provide a very cheap and efficient way to send a large number of payments and reduce the number of interactions with the blockchain.

Our goal is to design a protocol for executing micropayments nearly instantaneously. While the current blockchain mechanism is too expensive and slow to execute micropayment transactions, payment channels are a promising solution. However, while this technique nicely scales transactions between two parties, every new connection requires the setup of a new smart contract on the blockchain. To

1 Introduction

reduce this overhead, *payment networks*, like the lightning network [161], re-use existing channels and route payments via one or more intermediaries off-chain. For example, consider a scenario where party Alice has an open channel with Intermediary Ingrid, who, in turn, opened a channel with party Bob. This setup allows Alice to route a payment to Bob with Ingrid’s assistance. Alice promises to send the coins off-chain to Ingrid, but Ingrid will only be able to redeem this conditional payment if she sent the same sum to Bob through their channel. This setup allows Alice and Bob to send and receive payments off-chain without needing to trust Ingrid, who, at the same time, does not risk losing any funds. The problem with such previous constructions [161, 148, 165] is that they are inadequate for micropayments, as the routing over the intermediary adds additional delays and fees. The goal of this work is to find a new way of combining existing channels and allow the exchange of microtransactions without additional routing delays and costs.

To this end, we introduce PERUN, a protocol for *virtual payment channels*, that allows a faster and cheaper way of routing transactions. Virtual channels require the communication with untrusted intermediaries similarly to how direct channels use smart contracts: only during setup, closing, and in case of disputes. Since sending transactions in virtual channels does not require communication with the intermediaries, virtual payments are just as fast as payments through direct channels. We provide a detailed description of the PERUN protocol, formally model its functionality, and prove its security in the global Universally Composable (UC) framework [45]. Additionally, we analyze its efficiency with a prototype implementation in Ethereum and discuss extensions. In particular, we briefly describe how the system can be extended to provide not just payments but generic smart contracts [71] and how to build multi-party virtual channels [68].

Chapter 5: Moving Complex Computation Off-Chain. While Chapter 4 proposes a scaling solution that allows users to send simple payments off-chain, in Chapter 5, we aim to move complex smart contracts away from the blockchain. Our goal is to reduce execution costs for complex smart contracts without compromising on the security. This is achieved by keeping large inputs off-chain instead of sending them to the smart contract and by reducing the on-chain complexity of contract code. The FAIRSWAP protocol does not only improve the scalability and execution costs of complex contracts but also allows the evaluation of functions, which would be too large to execute directly in a smart contract. The protocol designed in this chapter is called FAIRSWAP, and it is based on the paper “Fair-

1 Introduction

Swap: How To Fairly Exchange Digital Goods” [67], which was presented at ACM Conference on Computer and Communications Security 2018.

In particular, we consider the case of a two-party smart contract, which runs a fair exchange of a (potentially large) digital commodity. This is the standard setting of a secure sale over the Internet, where a buyer is willing to pay a price of p coins to a seller if he receives a certain digital good x . This trade can be implemented naïvely by a smart contract, that takes p coins from the buyer and the commodity x from the seller. If x is “correct”, the contract transfers the money to the seller. In any other case, the money is sent back to the buyer. But in order for this to work, the contract needs to evaluate a predicate function ϕ , which outputs 1 if x is correct and 0 otherwise. Such a smart contract ensures financial fairness, which means that the price is only paid if and only if the buyer receives the correct x . However, if x gets large or the verification function ϕ is complicated, the contract gets too costly. Consider the example where x is a digital file that is identified by its publicly known hash h , i.e., the function ϕ outputs 1 only if the file x hashes to h . If x is a multimedia file, its size is easily in the range of gigabytes. A transaction that stores a single megabyte of data on the blockchain would cost approximately 319 euros in fees² and would not fit into a single transaction.

We propose an optimistic protocol called FAIRSWAP, which guarantees the same financial fairness as the straightforward solution and is efficient even for large x and/or complex ϕ . The idea is that the seller sends the encrypted x directly to the buyer, who will then lock the coins in the contract, thus confirming that he received the ciphertext. Only then will the seller reveal the key, which lets the buyer decrypt x . In case it is the expected file, the money goes to the seller, and the sale is successfully completed. However, in case the file is wrong, the buyer can prove this fact to the smart contract using only a (relatively) small statement, called *proof of misbehavior*. FAIRSWAP shows how both the size of this proof and its verification inside the contract can be kept very small in comparison to the size of ϕ and x . We also evaluate the efficiency of the scheme by providing a proof of concept implementation. We prove the security of this scheme by showing that if the file is wrong, the buyer can always expose the seller’s misbehavior. However, at the same time, a malicious buyer cannot produce such a statement if the file is correct, and the seller was honest. We additionally discuss extensions of the protocol [73], in particular how to make the proof of misbehavior interactive to make the optimistic case of the protocol even more efficient.

²We consider an exchange rate of 162.43 euros and a gas price of 3 GWei. More information on the Ethereum fee structure can be found in Section 3.2.1.

Chapter 6: Off-Chain Smart Contracts on Bitcoin. In this chapter, we investigate another direction of scaling complex smart contracts. Instead of running the contract on-chain or letting all parties execute the code locally (as in the previous proposals), we now outsource it to a Trusted Execution Environment (TEE) – a secure and trustworthy runtime environment for applications [159]. The resulting protocol is called FASTKITTEN and has been presented in the publication “FastKitten: Practical Smart Contracts on Bitcoin” [59] at the Usenix Security Symposium in 2019.

The main building block of this work is a TEE, such as Intel’s Software Guard Extension (SGX) [142, 99, 4], or the ARM TrustZone [7]. Such TEEs are specifically designed to be tamper-resistant, which allows them to run code in protected computation environments that strictly isolate a specific application on a potentially untrustworthy machine. The TEE guarantees confidential and correct code execution. We call the host or owner of the TEE the *operator*, and while the TEE itself is trusted, the operator can be malicious. This means he decides when to run the TEE, and he controls its inputs and outputs but cannot influence the computation inside the device or learn about its internal state. Based on the security of TEEs, we build the FASTKITTEN protocol, which guarantees efficient and fast evaluation of generic smart contracts that can interact with a fixed number of parties. Again, we minimize the interaction with the blockchain and only use it to lock the coins during the contract evaluation, and in the pessimistic case, to guarantee message delivery and penalize misbehaving players. Unlike the solutions in previous chapters, we show that FASTKITTEN only requires simple transactions that are supported by blockchains without advanced scripting capabilities. We provide a construction that works on Bitcoin and analyze its efficiency and fees. We formally prove the security of the scheme and show that malicious parties cannot influence the computation of the contract or steal coins from it.

1.4 Structure of this Thesis

In Chapter 2, we introduce all cryptographic building blocks that we use in the protocols, i.e., hash functions, encryption, commitment, and signature schemes. We present the formal notation and definitions that we rely on in this thesis. Additionally, we discuss the concept and essential aspects of provable security and present the relevant formal models.

Chapter 3 gives a detailed introduction and explanation of blockchain technology and smart contracts. We also analyze their limited transaction throughput

1 Introduction

and compare previous proposals that aim at improving scalability. Additionally, we focus on the formal properties of cryptocurrencies. In particular, we discuss the trust assumptions needed to build distributed ledgers and what security they can offer and present how we formally treat the handling of coins and the communication with the blockchain.

In the following three chapters, we discuss the protocols PERUN, FAIRSWAP, and FASTKITTEN, that have been developed in this thesis. The content of each chapter follows the same order. The contribution is summarized for each of them at the beginning. Then, we provide a more detailed overview, which presents the motivation and high-level design ideas for each scheme and discusses additional related work. A preliminary section in each chapter briefly presents additional formal notations and models, if necessary. Next, we define the security and construct the protocol. Following this, we present the security proof, and the benchmarking results from our implementation. Finally, we discuss the results and present extensions to the initial protocol design.

In the final Chapter 7 of this thesis, we compare the three proposed protocols from the previous chapters and discuss their advantages and shortcomings. We give an overview of what applications can benefit from their usage.

2 Cryptographic Preliminaries

In this chapter we introduce the basic notions and formal cryptographic definitions that we will use in this thesis.

2.1 Notation

We denote the set of natural numbers $1, \dots, m$ as $[m]$, the set of all binary strings with the length of n bits as $\{0, 1\}^n$, the set of all bit strings as $\{0, 1\}^*$, and the n bit string comprised only of 1s as 1^n . Whenever we consider a probabilistic algorithm A , then $y \leftarrow A(x)$ denotes that the output y is generated by A using internal randomness r . For deterministic algorithms or whenever we make this internal randomness explicit we write $y := A(x, r)$ instead. When a value x is sampled uniformly at random from a set X we write $x \xleftarrow{\$} X$. We will use κ to denote security parameters, and say that a function negl is negligible if for all positive polynomials poly there exists some constant κ_0 such that for all $\kappa > \kappa_0$ it holds that $\text{negl}(\kappa) < \frac{1}{\text{poly}(\kappa)}$.

We denote algorithms Alg with upper case serif free fonts and parties \mathcal{Q} with calligraphic letters. Parties are modeled as interactive Probabilistic Polynomial Time (PPT) Turing machines. Whenever parties send messages to each other, we will give each message a name, denoted with typewriter font, i.e., a message with value x is denoted as $(\text{msgName}, x)$.

The notion of computational indistinguishability [108] is a heavily used in modern cryptography. We say two distributions \mathbf{X} and \mathbf{Y} are *computationally indistinguishable* if it is difficult to tell them apart (cf. formal Definition 1). We write $\mathbf{X} \approx_c \mathbf{Y}$.

Definition 1 (Computational indistinguishability). *Let κ be a security parameter. Two distribution ensembles $\mathbf{X} = \{X(\kappa)\}_\kappa$ and $\mathbf{Y} = \{Y(\kappa)\}_\kappa$ are computationally indistinguishable if for every PPT distinguisher \mathcal{A} there exists a negligible function negl s.t.:*

$$\left| \Pr[\mathcal{A}(1^\kappa, x) = 1] - \Pr[\mathcal{A}(1^\kappa, y) = 1] \right| \leq \text{negl}(\kappa)$$

2 Cryptographic Preliminaries

where the probability is taken over the randomness of algorithm \mathcal{A} and the random sampling of $x \xleftarrow{\$} X$ and $y \xleftarrow{\$} Y$.

When we define security, we often consider experiments, that an algorithm \mathcal{A} is trying to win. We will argue about the winning probability of \mathcal{A} , where winning requires him to run the (randomized) experiment and produce a specific event. We denote this probability as

$$\Pr[\text{event} : \text{experiment}]$$

where the probability is always taken over the randomness of that experiment.

2.2 Provable Security

The concept of provable security is used to formally argue about the security of cryptographic algorithms and protocols. The paradigm of the field of modern cryptography is to apply rigorous logical argumentation in the form of mathematical proofs, to show that the analyzed schemes cannot be attacked. Before such a proof can be conducted, the following has to be specified.

Adversary model: We need to describe any possible attacker that our protocol should protect against. Modeling this attacker includes a clear description of its power and capabilities. In this thesis we consider *computational security*, which models the adversary as a poly-time bounded¹ algorithm which is allowed to have a negligible probability of breaking the security (the attackers advantage). This means that the adversary can break the security with a negligible probability, i.e., by guessing or brute-force.

Formal definitions: The first step to a security analysis of every cryptographic scheme is a sound formal definition of its security. Such a definition captures the required properties of the scheme by defining the threat model. This is usually done by providing the adversary model, which defines the capabilities that an attacker has, i.e., what he can and what he cannot do. The goal of the adversary is formulated either in a game based or in a simulation based fashion. In game based security, a security property of the scheme is

¹In particular we consider algorithms which require runtime that is polynomial in the security parameter and are successful with probability which is negligible in the same security parameter.

2 Cryptographic Preliminaries

defined and analyzed over an adversary who has to win a predefined game or experiment in order to break the analyzed security property. We say the system is secure if the adversaries' advantage (the probability of him winning the game) is at most negligible in the security parameter. In simulation based security definitions the adversary needs to distinguish between the original and a simulated version of the security experiment and his advantage describes the probability which with he wins the experiment apart from guessing.

Assumptions: The security of most cryptographic schemes relies on assumptions. Typically examples for such assumptions are that some mathematical problem is hard to solve (e.g., factorization of large numbers) or on the existence of cryptographic primitives and their security. These assumptions should be easy to state and must be well studied by the cryptographic community.

Security proofs are often non-trivial as we need to show that there does not exist an adversary that is able to break the stated security definition. Therefore, showing security against specific attackers would not be helpful in these proofs. Instead, we utilize *reductions* (a common proof technique of modern cryptography) to show that the existence of an adversary would contradict our assumptions. We will often reduce the security of a scheme to the security of one of the primitives that we use. Thus, often security statements define a security property of a scheme in relation to the underlying assumptions.

Security of cryptographic protocols. In this thesis, we design and analyze *interactive cryptographic protocols* between two or more parties. At the beginning of the protocol, parties get input, and at the end, they output some value. In general, we consider a static adversary that can choose which parties to *corrupt* but only before the start of the protocol. A corrupted party is controlled by the adversary, who learns all its inputs and decides how it behaves. If the adversary possesses additional powers, they must be clearly defined.

In cryptography, we can choose from different models for conducting security proofs. In particular, we distinguish between the *standalone model* and the *Universally Composable (UC) model* in this thesis. In the standalone model, we analyze the security of isolated single protocol execution. For this purpose, we define every security property of the protocol in the presence of a PPT adversary who tries to break this property. In Chapter 6, we follow this model and prove the security properties for a standalone protocol execution.

2 Cryptographic Preliminaries

The main limitation of the standalone model is that it ignores a much more complicated world where multiple protocol instances are executed by many parties at the same time. To address this shortcoming, a different model for proving security of cryptographic protocols, called the Universally Composable (UC) framework, was proposed by Ran Canetti in 2000 [43]. The UC model captures the security of concurrent (parallel and sequential) protocol executions between many parties and even in composition with other protocols. In contrast to the standalone approach, protocols are analyzed by running them in the presence of an environment that initiates the protocol executions, selects all inputs, and controls the adversary’s behavior (and that of corrupted parties). Security in this model is defined through an *ideal functionality*, which captures the ideal outcome of the protocol. This ideal representation defines all security properties that the final protocol should have. Additionally, it models the inputs and outputs of every party, and all leakage and influence that each of them gets (including the adversary). The security proof then analyzes the differences in the execution of the ideal functionality and the protocol, often called the real-world. The key idea is that if these executions cannot be distinguished (by the environment), then the protocol is just as secure as the ideal representation. We provide a more formal overview of the UC framework and its components below in Section 2.4, as this model is used in Chapters 4 and 5.

2.3 Cryptographic Primitives

Next, we present the basic cryptographic primitives that are required for the rest of this thesis. In particular, we define commitment, signature and encryption schemes as well as hash functions and their security.

Hash Functions

Cryptographic hash functions are important cryptographic primitives, which are heavily used by the schemes presented in this thesis as well as for cryptocurrencies in general. A hash function H (as considered in this thesis) maps binary strings of arbitrary length to binary strings of a fixed length μ . Formally, hash functions are modeled as keyed hash functions, where a key generation algorithm Gen , which takes as input the security parameter κ , selects a seed from a key-space K . This seed is used to parameterize the deterministic hash function H^k . We require that the hash function H^k satisfies collision resistance [167] for a sufficiently large parameter μ .

2 Cryptographic Preliminaries

Definition 2 (Collision Resistant Hash Functions). *Let κ be a security parameter. A keyed hash function is a tuple of algorithms (Gen, H^k) where Gen on input of 1^κ outputs a seed $k \in K$ and H^k indexed by the seed k outputs a hash value $h \in \{0, 1\}^\mu$ on input of a value $v \in \{0, 1\}^*$. A keyed hash function is collision resistant, if for any PPT adversary \mathcal{A} there exists a negligible function negl s.t.:*

$$\Pr[H^k(v) = H^k(v') \text{ AND } v \neq v' : k \leftarrow \text{Gen}(1^\kappa), (v, v') \leftarrow \mathcal{A}(k)] \leq \text{negl}(\kappa)$$

where the probability is taken over the randomness of algorithms Gen, H^k , and \mathcal{A} .

In this thesis we will follow the convention of writing H instead of H^k as it simplifies the exposition of hash functions. In Chapter 5 we will model collision resistant hash functions as random oracles, which we will introduce in more detail in Section 5.2.

Merkle Trees Merkle trees [144], provide domain-extension for collision resistant hash functions. In particular, they allow to hash a large input string to a constant sized digest $h \in \{0, 1\}^\mu$. This is particularly useful when many (say n) elements are hashed to a single value, as it is possible to generate a short proof that a certain element is part of the tree. The key idea is to create a complete binary tree, which has the n values as leaves, and every node is a hash of its children. The root of that tree serves as a digest h of the n values, and whenever it needs to be shown that a particular element was part of the tree, we open all values (i.e., the siblings) on the path between the node and the root hash, resulting in a number of $\lceil \log(n) \rceil$ elements. In Section 5.2, we provide more details on how to construct Merkle trees using the algorithms $\text{Mtree}^{\mathcal{H}}$, $\text{Mproof}^{\mathcal{H}}$, and $\text{Mvrfy}^{\mathcal{H}}$.

Encryption Schemes

A symmetric or private key encryption scheme allows to encrypt a message x from a message space X under a secret key k , such that the ciphertext without the key does not reveal information about x . Formally, a symmetric encryption scheme for a message space X consists of three PPT algorithms $(\text{Gen}, \text{Enc}, \text{Dec})$. The key generation algorithm Gen takes as input 1^κ , where κ is the security parameter, and outputs a key k from key space K . The encryption algorithm Enc takes as input a key $k \in K$ and a message $x \in X$ and outputs a ciphertext c from ciphertext space C . The deterministic decryption algorithm Dec , which takes as input the key $k \in K$ and ciphertext $c \in C$, outputs x or an error denoted as \perp .

2 Cryptographic Preliminaries

We require correctness of the scheme, i.e., that for all $k \leftarrow \text{Gen}(1^\kappa)$ and $x \in X$: $\text{Dec}(k, \text{Enc}(k, x)) = x$.

In contrast to symmetric encryption schemes, asymmetric or public key encryption schemes allow anyone to encrypt a message x under a known public key pk . But only the owner of the corresponding secret key sk , can decrypt the ciphertext. Formally, an asymmetric encryption scheme for messages $x \in X$ consists of three PPT algorithms $(\text{Gen}, \text{Enc}, \text{Dec})$. The key generation algorithm Gen takes as input 1^κ , where κ is the security parameter, and outputs a key pair $(sk, pk) \leftarrow \text{Gen}(1^\kappa)$. The encryption algorithm Enc takes as input a public key pk and a message $x \in X$ and outputs a ciphertext $c \in C$. The deterministic decryption algorithm Dec , which takes as input a secret key sk and a ciphertext $c \in C$ outputs x or an error denoted as \perp . We require correctness of the scheme, i.e., that for all $(sk, pk) \leftarrow \text{Gen}(1^\kappa)$ and $x \in X$: $\text{Dec}(sk, \text{Enc}(pk, x)) = x$.

The asymmetric encryption scheme used in this thesis needs to be *indistinguishable under chosen plaintext attacks* (IND-CPA security) [108]. This means that for any PPT adversary that chooses two messages x_0, x_1 (of the same length) and learns $c = \text{Enc}(pk, x_b)$ for a randomly chosen bit b , it must be hard to guess b correctly except with negligible advantage. The adversary in this case is called twice, once to provide two inputs x_1, x_2 and a second time to choose which of the two inputs was encrypted. Note, that the adversary stores a state between these two calls and (as it gets the public key as input) can encrypt arbitrary messages.

Definition 3 (IND-CPA Secure Encryption). *Let κ be a security parameter. A public key encryption scheme $(\text{Gen}, \text{Enc}, \text{Dec})$ has is indistinguishable ciphertexts under chosen plaintext attacks (is IND-CPA secure) if for all PPT adversaries \mathcal{A} there exists a negligible function negl s.t.*

$$\Pr[b = b' : (sk, pk) \leftarrow \text{Gen}(1^\kappa), (x_0, x_1) \leftarrow \mathcal{A}(1^\kappa, pk), \\ b \xleftarrow{\$} \{0, 1\}, c \leftarrow \text{Enc}(pk, x_b), b' \leftarrow \mathcal{A}(c)] \leq \frac{1}{2} + \text{negl}(\kappa)$$

where the probability is taken over the randomness of algorithms Gen, Enc and \mathcal{A} and the random choice of $b \xleftarrow{\$} \{0, 1\}$.

Digital Signature Schemes

Digital signatures allow a sender to authenticate messages with a secret key sk , such that every recipient with knowledge of the corresponding public key pk can be ensured that the message was sent by the sender. In particular, it is computationally infeasible to generate a valid signature without the secret key. Formally,

2 Cryptographic Preliminaries

a signature scheme for messages from a message space X is a triple of PPT algorithms $(\text{Gen}, \text{Sign}, \text{Vrfy})$. The key generation algorithm Gen takes as input 1^κ , where κ is the security parameter, and outputs a key pair (sk, pk) from key space K . The signature algorithm Sign takes as input a secret key sk and a message $x \in X$ and outputs a signature σ . The deterministic verification algorithm Vrfy takes as input the public key pk , the message x , and the signature σ and outputs 1 if the signature is correct or 0 otherwise. Again, we require correctness of the scheme, i.e., that for all $(sk, pk) \leftarrow \text{Gen}(1^\kappa)$ and $x \in X$: $\text{Vrfy}(pk, x, \text{Sign}(sk, x)) = 1$. We require that the signature scheme is existentially unforgeable against adaptive chosen message attacks (we will also say EUF-CPA secure). This means we protect against an adversary who can see valid signatures for messages of his choice. Formally this is modeled as a signing oracle which on input of a value x outputs the signature $\sigma = \text{Sign}(sk, x)$. The task of the adversary is to produce a fresh message (that has not been queried before to the signing oracle) and a valid signature with respect to pk .

Definition 4 (Unforgeability Against Adaptive Chosen Message Attacks). *Let κ be a security parameter. A signature scheme $(\text{Gen}, \text{Sign}, \text{Vrfy})$ is existentially unforgeable against adaptive chosen message attacks if for all PPT adversaries $\mathcal{A}^{\text{Sign}}$ with oracle access to the signing oracle $\text{Sign}(sk, \cdot)$, there exists a negligible function negl s.t.*

$$\Pr[\text{Vrfy}(pk, x, \sigma) = 1 \text{ AND } x \notin Q : (sk, pk) \leftarrow \text{Gen}(1^\kappa) \\ (x, \sigma) \leftarrow \mathcal{A}^{\text{Sign}(sk, \cdot)}(1^\kappa, pk)] \leq \text{negl}(\kappa),$$

where Q is the set of all queries that \mathcal{A} made to the signing oracle and the probability is taken over the randomness of algorithms Gen, Sign and \mathcal{A} .

Commitment Schemes

A commitment scheme allows a sender to convince a receiver that he fixed (or committed to) a message x by providing a commitment c , which does not reveal information about x . At a later point he can reveal x and prove that it was committed to by providing an opening value d . Formally, a commitment scheme for input values $x \in \{0, 1\}^*$ consists of three algorithms $(\text{Gen}, \text{Commit}, \text{Open})$, where the key generation algorithm Gen takes as input 1^κ , where κ is the security parameter, and outputs public parameters pp . The (probabilistic) algorithm $\text{Commit}_{pp}(x)$ is parameterized by these public parameters and outputs a commitment c and an

2 Cryptographic Preliminaries

opening value d , and the algorithm $\text{Open}_{pp}(c, d) = x$ outputs x for a valid commitment $(c, d) \leftarrow \text{Commit}(x)$ and \perp otherwise. A commitment scheme is correct, if for all $x \in X$: $\text{Open}_{pp}(\text{Commit}_{pp}(x)) = x$ with $pp \leftarrow \text{Gen}(1^\kappa)$. Cryptographically secure commitment schemes [108] have to satisfy the *hiding* and *binding* properties. Hiding guarantees that for any two messages x, x' and $(c, d) = \text{Commit}_{pp}(x)$ and $(c', d') = \text{Commit}_{pp}(x')$, we have that $c \approx_c c'$.

Definition 5 (Computationally Hiding Commitments). *Let κ be a security parameter. A commitment scheme $C = (\text{Gen}, \text{Commit}, \text{Open})$ is hiding, if for any PPT adversary \mathcal{A} there exists a negligible function negl s.t.:*

$$\Pr[b = b' : pp \leftarrow \text{Gen}(1^\kappa), (x_0, x_1) \leftarrow \mathcal{A}(1^\kappa, pp), b \xleftarrow{\$} \{0, 1\}, \\ (c, d) \leftarrow \text{Commit}_{pp}(x_b), b' \leftarrow \mathcal{A}(c)] \leq \frac{1}{2} + \text{negl}(\kappa)$$

where the probability is taken over the randomness of Gen , Commit and \mathcal{A} .

The binding property prevents the committer from being able to open the commitment to a different value than x . In particular, it requires that it is computationally hard for any PPT adversary \mathcal{A} to find a triple (c, d, d') such that $\text{Open}_{pp}(c, d) = x$ and $\text{Open}_{pp}(c, d') = x'$ with $x \neq x'$ and $x, x' \neq \perp$.

Definition 6 (Computationally Binding Commitments). *Let κ be a security parameter. A commitment scheme $(\text{Gen}, \text{Commit}, \text{Open})$ is binding, if for any PPT adversary \mathcal{A} there exists a negligible function negl s.t.:*

$$\Pr[\text{Open}_{pp}(c, d) = x \text{ AND } \text{Open}_{pp}(c, d') = x' \text{ AND } x \neq x' \text{ AND } x, x' \neq \perp : \\ pp \leftarrow \text{Gen}(1^\kappa), (c, d, d') \leftarrow \mathcal{A}(1^\kappa, pp)] \leq \text{negl}(\kappa)$$

where the probability is taken over the randomness of Gen , Commit , and \mathcal{A} as well as the random choice of $b \xleftarrow{\$} \{0, 1\}$.

For simplicity we will often omit the public parameters and write Commit and Open without explicitly mentioning pp .

2.4 The Universal Composability Model

One common method to describe and analyze complex cryptographic protocols is the universal composability (UC) framework of Canetti [43]. We analyze a protocol Π which runs among a set of parties P , that are modeled as interactive

2 Cryptographic Preliminaries

PPT Turing machines. In the UC framework, security of a protocol is analyzed by comparing its execution with an idealized and simplified trusted protocol execution \mathcal{F} , which we call ideal functionalities. The case when the parties interact with the real protocol Π is called the *real world* execution while the *ideal world* means the parties interact with the ideal functionality \mathcal{F} instead. Both worlds are operated by a special party \mathcal{Z} – the so-called environment, which selects inputs for the protocol participants and receives their outputs. In the *real world* Π is executed among the set of parties P , which are connected by authenticated communication channels that guarantee delivery of messages within one round. In addition, a special party called the *adversary* \mathcal{A} may corrupt parties. Corruption of a party $\mathcal{P} \in P$ means that the adversary takes full control of \mathcal{P} 's actions and learns his internal state. For simplicity, we consider a static adversary, where corruption only takes place at the beginning of the protocol. Formally, the output of the real world execution of protocol Π with input x is denoted as

$$REAL_{\Pi}^{\mathcal{Z}, \mathcal{A}}(\kappa, x).$$

To analyze the security of the protocol Π in the real world, we compare its execution with an idealized protocol execution. In the *ideal world* we consider a dummy protocol where the parties from set P just forward their inputs to an ideal functionality \mathcal{F} . We call these parties dummy parties. The ideal functionality specifies the protocol's interface and can be viewed as an abstract specification of what security properties Π shall achieve. In the ideal world, the ideal functionality can be attacked through its interface by an ideal world adversary – called the simulator *Sim*. Formally, we denote the output of the ideal world execution as

$$IDEAL_{\mathcal{F}}^{\mathcal{Z}, Sim}(\kappa, x).$$

The environment \mathcal{Z} orchestrates both worlds by providing the inputs for all parties, and receiving their outputs. But this party acts as a distinguisher, which means that it does not know which of the worlds it is interacting with. We say a protocol Π is *UC-secure* if the environment \mathcal{Z} cannot distinguish whether it is interacting with the ideal or real world. This indistinguishability must hold for all PPT environments \mathcal{Z} . We prove this security formally by showing that for every real world adversary \mathcal{A} we can construct an ideal world simulator *Sim* such that for all environments \mathcal{Z} and all inputs x the following holds:

$$REAL_{\Pi}^{\mathcal{Z}, \mathcal{A}}(\kappa, x) \approx_c IDEAL_{\mathcal{F}}^{\mathcal{Z}, Sim}(\kappa, x).$$

2 Cryptographic Preliminaries

Security in the UC framework implies that a protocol Π is at least as secure as the ideal functionality \mathcal{F} . We will then also say that a protocol *emulates* the ideal functionality.

We use a particularly useful property of the UC framework to modularize the design of our protocols. The *universal composition theorem* [43] allows us to use ideal functionalities as subroutines in our protocols and later replace them by respective protocols that realizes these functionalities. A protocol which uses an ideal functionality \mathcal{F} as a sub-routine is often said to run in the \mathcal{F} -hybrid world. We will apply this technique to model interactions with smart contracts (cf. Section 3.2). To this end, we define a hybrid world where the protocol has access to an idealized smart contract functionality \mathcal{C} . The execution of the functionality is trusted and can only be influenced through its specified interface. If a hybrid functionality is used during the protocol execution we denote the output of this hybrid-world protocol execution as

$$HYBRID_{\Pi, \mathcal{C}}^{\mathcal{Z}, \mathcal{A}}(\kappa, x)$$

and show that it is indistinguishable from the output of the ideal world execution.

Global UC model. A shortcoming of the UC framework is that – even though it models concurrent protocol executions – it does not allow that different executions share the same hybrid functionalities. This, however, leads to unwanted restrictions when it comes to modeling a protocol component (like a random oracle [46] or a common reference string [45]) that is available in many different protocols. In the traditional UC model, security could only be guaranteed if, for every single protocol instance, a different and independent component is used. However, this does not correspond to how these components are implemented in practice. E.g., the random oracle is often replaced with the same hash function. To overcome this limitation, in [45], the authors propose an extension to the UC framework, which allows for *global shared functionalities*. In particular, these functionalities can be accessed from both the ideal and real-world execution and store global state.

In Chapters 4 and 5, we will use global functionalities to model the functioning of the ledger, which (for concurrent protocol executions) handles the secure coin transfer, as realized by blockchains. By modeling the ledger as a global functionality, we allow that our smart contracts (as hybrid functionalities) make changes on the ledger. These changes affect all concurrent protocol executions and are public to any observer. This modeling ensures that our protocols are secure, even if the balances on the ledger are affected by other concurrent executions of the same or

2 Cryptographic Preliminaries

different protocols. In Chapter 5, we will additionally model the random oracle as a global functionality following the work of [41]. We provide detailed information on the model we use in Section 5.2.

2 *Cryptographic Preliminaries*

3 Blockchain Technology

Since the invention of the Internet, researchers (i.e., [49, 50, 32]) envisioned digital cash systems that allow everyone to own, transfer, and receive money in a decentralized and trustless manner. Instead, only a handful of large companies like Visa, Mastercard, or Paypal offer the financial infrastructure that allows monetary transfers all over the globe. But all of these providers are controlled by a central authority that the users have to trust. These companies control which users and transactions to accept and how the transfers are processed. Additionally, these companies benefit from monitoring and storing the transaction history to create individual customer profiles.

In 1985 David Chaum proposed anonymous electronic cash (in short e-cash) [49, 50], which was later extended by Brands [32] and many others. While there is still a central bank in these proposals, it issues coins anonymously. The users withdraw these coins and can spend them to any merchant they like. When the merchant deposits the coins back to his bank account, he has the guarantee that these coins were valid, but the Bank cannot link the user to the merchant. While e-cash systems in some regards mimic the traditional cash systems, they still require trust into a central component, which decides on exchange rates and who can withdraw coins.

With the proposal of Bitcoin in 2008 [151] by Satoshi Nakamoto, a new digital currency was invented that does not rely on a trusted intermediary but distributes the trust to a decentralized network instead. The challenge of Bitcoin is to achieve some form of *Consensus* within this network. However, in contrast to traditional consensus [127], the set of parties in the network is unknown and may change over time. This setup is often referred to as the *permissionless model* of consensus.

In this chapter, we will first present the basic building blocks and mechanics of Bitcoin. We will also present another cryptocurrency, called Ethereum, and their main feature smart contracts. While the results in this thesis could work on other currencies as well, these two currencies will serve as exemplary technologies for the modeling and implementation, since they are the largest of their kind. In Section 3.4, we present an overview of the formal security properties that

blockchain-based cryptocurrencies like Bitcoin and Ethereum provide.

3.1 Bitcoin

We start by describing the original Bitcoin protocol, as described in [151]. There are many challenges in designing a secure, distributed currency. It must guarantee that only the owner of a coin can decide when and how to transfer a coin, and at the same time, prevent that someone spends a coin twice (double spending). Creating a unified global decision on the status of a transaction is difficult in distributed systems where the number of participants is unknown, and the creation of new nodes is cheap. In such distributed networks, we must always assume that some participants have malicious intent. They might for example want to change the validity of transactions in retrospect for their enrichment or censorship. In this section, we describe how Bitcoin solves these challenges step by step, starting with the layout and mechanics of the used data types, i.e., the transactions and blocks. Afterwards, we explore how these elements are used in the overall Bitcoin protocol and how the security of the system is guaranteed.

3.1.1 Transactions

Digital signatures¹ secure the ownership and correct transfer of coins (cf. Section 2.3). Bitcoin users are identified via their *addresses*, which is (a hash of) their public signing key. When a user wants to spend his coins, he specifies a receiver using his address and authorizes the payment by providing a digital signature. Therefore digital signatures ensure that only the owner of the corresponding secret key can *create* a transaction. Once a transaction is signed, however, anyone can publish it by sending it to the network. Bitcoin transactions work in the so-called Unspent Transaction Output (UTXO) Model.

Transactions are the fundamental underlying data structure of Bitcoin. Every transaction tx contains a list of *input references*, and a list of *output scripts*. Every input reference points to a prior transaction (more specifically to one of its output scripts), which will be spent by transaction tx and a witness. Output scripts specify the rules on how to redeem a certain number of coins in the Bitcoin scripting language. If a transaction distributes the money to multiple destinations, it requires multiple output scripts. A transaction can also contain multiple inputs and

¹The underlying signature scheme used in Bitcoin is the Elliptic Curve Digital Signature Algorithm (ECDSA).

combine coins from different sources. To spent transaction tx_1 (i.e., refer to it as in the input of a later transaction tx_2), the spender has to append a correct witness, often in the form of a signature. Miners accept a transaction tx_2 that spends an output of tx_1 if the output script of tx_1 in combination with the transaction witness of tx_2 evaluates to **true**. Most (standard) output scripts that send coins to another party, only specify its address, i.e., the hash of the public key of the receiver. In this thesis, we mainly require these simple standard **pay-to-pubkey-hash** transactions, which are redeemed using the signature of the sender. Additionally, we need two more features of the scripting language. The **OP_Return** [151] instruction allows storing data in a transaction and the **OP_CheckTimeLockVerify** [178] instruction timelocks it. The later allows us to specify at which future time² a transaction will be considered valid. Similar to [5], we represent transactions by tables, as shown exemplary in Figure 3.1.

Transaction tx	
tx.Input:	Coins from unspent input transaction
tx.Output:	Coins to receiver address
tx.Time:	Some timelock (optional)
tx.Data:	Some data (optional)

Figure 3.1: A simple transaction tx with a single input and output.

Bitcoin also supports more complex transaction outputs as long as they are stated in the minimalistic Bitcoin scripting language (SCRIPT). This language supports simple expressions like boolean arithmetic and enables constructions such as fair commit-reveal schemes [23]. Several works have shown how Bitcoin can support lotteries [147], poker games [123], and generic MPC [122] protocols. Other cryptocurrencies provide more expressive scripting capabilities. In Section 3.2.1 we present Ethereum transactions which support Turing complete instructions – also called *smart contracts*.

3.1.2 Blocks and Mining

All participants of the Bitcoin network must agree whether a transaction is valid, which requires that they agree on the order of transactions. This ordering is achieved by sorting them into a list of blocks, the so-called *blockchain*. Every Bitcoin block includes an ordered list of valid transactions. New blocks are proposed

²Time is measured in total block count.

3 Blockchain Technology

on average every 10 minutes and extend the tail of the chain by referencing the previous block (via its hash). The hash references link all blocks together, starting with the first, so-called genesis block. They ensure that any attacker, who wants to change a single block (or its content) needs to change the whole chain from that point onward.

The blockchain is stored redundantly by every node in the distributed Bitcoin network. Some of these nodes, called *miners*, work on the creation of new blocks. They collect new transactions, verify their correctness, and try to publish them in new blocks. Together, all miners secure the correctness and liveness of the cryptocurrency [85]. By verifying and confirming new blocks, they ensure that no false blocks are published and by proposing new blocks, they guarantee that new (valid) transactions are included in the blockchain eventually (more formally these properties are discussed in Section 3.4).

The underlying network is open, which means everyone can join as a new node and can become a miner. The peers are connected through a *gossip network*, where new transactions and blocks propagate by being forwarded from one node to its neighbors.

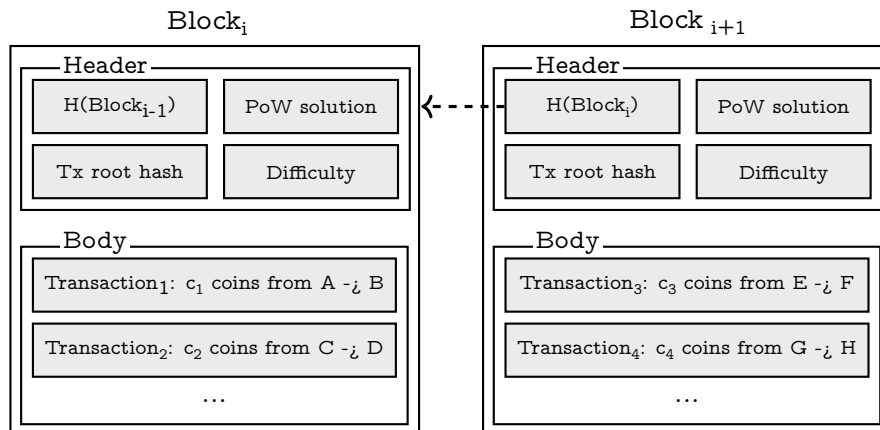


Figure 3.2: Bitcoin Blockchain

The open and unregulated nature of these systems raises the need for protection against *Sybil attacks* [65]. In traditional voting style systems, attackers can gain an unfair advantage from creating many fake identities and increasing their influence on the vote. In Bitcoin, this problem is prevented by binding the miners' influence on the computational resources that they are willing to invest.

Proof of Works. A Bitcoin block consists of a header and a body that includes the transactions (cf. Figure 3.2). The header contains, among other values, the reference to the previous block, a hash of all transactions in the body (i.e., a Merkle tree root hash).

Every block header must also include the solution to a so-called PoW, a puzzle that requires a significant amount of computational power to solve. The purpose of these schemes is that a prover can convince a verifier that he spent (on average) a certain amount of work on finding a solution. The proof must be easy to verify, and its difficulty should be adaptable. PoWs have been proposed in the literature before, i.e., for denial-of-service or spam prevention [12]. In Bitcoin, PoWs prevent Sybil attacks, as they ensure that the miners' contribution to finding new blocks is proportional to their computational power. The mining process works as follows: A miner (acting as the prover) has to provide a value which (in combination with the current block header and the hash of the latest block) hashes to a sufficiently small number (the difficulty). The best strategy to find this PoW solution is guessing, which means that all miners race against each other to find the solution first and publish the next block. The probability of winning this race increases with the number of computational resources a miner invests in this task.

Whenever a new block is found, it is sent to all miners and the race restarts. The case where there are two (or more) blocks competing against each other is called a *fork*. This case can happen, when two miners find a block at the same time or when a malicious miner publishes another version of a recent block. Honest miners reject blocks that include invalid transactions, and malicious miners might propose alternative blocks to change the transaction order. While, in rare cases, forks can occur for the duration of a few blocks, they are usually quickly resolved. Honest miners will always choose the longest (or rather the most difficult) fork and try to extend it, which guarantees that all honest miners will work on the same branch of the chain eventually, and the other branch will perish. The temporary risk of forks leads to the recommendation to wait a while until a block is considered valid. In fact, it is said only to accept a block which has been confirmed at least by six newer blocks.

To keep the average delay between blocks fixed (to on average of 10 minutes) the difficulty is adapted in regular intervals to account for changes in the invested mining power. At the time of writing this thesis, the hashing power which is invested in Bitcoin is equal to approximately $1,153 \times 10^8$ THash per second.

3.1.3 Bitcoin Fees

The primary reason for miners to invest in and spent energy on running mining hardware is because it is profitable. For every new block, that a miner publishes, he gets a block reward and transaction fees.

Mining rewards. The block reward consists of newly minted coins, which are created according to the rules of the protocols and can be claimed by the miner of the block. When the Bitcoin currency started in 2009, the block reward was set to 50 BTC. As the value of coins rises over time and to counter inflation, the reward halves every 210 000 blocks as more and more coins are created. The overall Bitcoin supply will stagnate at around 21 million coins in the year 2140 [151].

Transaction Fees. In addition to the mining reward, the miner will also get the transaction fees inside the block. The sender of a transaction sets and pays the transaction fees, which means he can freely choose how much he is willing to pay (if at all). As the size of blocks is limited by 1 megabyte, the individual fees per byte influence which transactions are most profitable for miners and will most likely be included first. Table 3.1 gives an overview of how Bitcoin fees influence the confirmation speed. Transaction fees in Bitcoin are typically stated in satoshi (the lowest denomination of Bitcoin equal to 1×10^{-8} BTC) per Byte.

Priority	Fee per Byte	Waiting Time	# of Blocks
low	1 satoshi	300 min	19
medium	5 satoshi	180 min	9.5
high	10 satoshi	55 min	4
very high	15 satoshi	35 min	1

Table 3.1: Average time and number of blocks that it takes until a Bitcoin transaction is included (numbers from [27])

In this thesis we will assume transactions have a very high priority, thus they should be included in the next block (by an honest miner). This means we consider transaction fees around 15 satoshi per byte. The median size of Bitcoin transactions is 214 bytes, which means we consider transaction fees around 0.26 euros per transaction. We assume for this calculation an exchange rate of 7951.95 euros per BTC, which corresponds to the 180 day average exchange rate

of Bitcoin (calculated from the 24th of February 2020). The data was taken from blockchain.com [28].

3.2 Ethereum and Smart Contracts

The second biggest cryptocurrency is Ethereum, which was proposed in 2014 by Vitalik Buterin [37] and later formalized [186] and implemented. The design of Ethereum differs from Bitcoin in some ways but the most important one is that it provides rich scripting features and *smart contracts*. It enables countless novel applications and is often regarded as a glimpse into our future.

3.2.1 Smart Contracts

Informally speaking, smart contracts are coded agreements, which are stored on the blockchain, that can receive, store, and redistribute coins depending on some well-specified conditions. Smart contracts bind money transfers to program code, and thereby allow to execute transactions based on complex contractual agreements enforced by the miners of the cryptocurrency. Unlike Bitcoin, Ethereum does not work in the UTXO model but in an account-based model, which distinguishes two types of accounts:

Externally owned accounts Represented though a public address, based on an ECDSA public key, these accounts are controlled by users. Sending coins requires a correctly signed transaction.

Contract accounts These accounts are controlled by their contract code, which describes how stored coins are redistributed. Contract addresses are generated during deployment.

Every account is identified over an address and can hold Ether (the currency unit in Ethereum). Contract accounts additionally also store their contract code (in bytecode form) and storage. Once deployed, the contract is public and all users may interact with it. A smart contract cannot act on its own. Instead, it needs to be triggered by a transaction from an externally owned account. Such a contract function call contains the function parameters and might optionally contain funds that are sent to the smart contract. Such a transaction is executed by the miners that evaluate the contract function code with the provided parameters as input. As a result, the contract state is updated. All honest miners verify the correctness

of published blocks and, thus, also of the state updates. All miners must have the same view on the code and execution of contracts. Thus they can only run deterministic code and cannot have a private state.

Smart contracts are passive pieces of code that do not act on their own nor interact with users directly. Therefore, we require that whenever a contract function needs to be evaluated, a user triggers the contract with a function call. If one contract is activated from an externally owned account, it can also call other contracts via their address. A call from one contract \mathcal{C}_A to another contract \mathcal{C}_B is called a contract message. They work similarly to contract calls and reference a specific function. Contract messages can contain function parameters and even transfer coins. After the function evaluation in contract \mathcal{C}_B finishes, the rest of the contract \mathcal{C}_A is executed.

Miners locally store the state of the Ethereum system, which contains the current state of all active contracts. Ethereum blocks contain both the list of transactions and a hash of the most recent state. This allows Ethereum nodes to execute contract calls quickly and verify the state transition proposed by new blocks.

Contract Deployment

In Ethereum, smart contracts can be written in a scripting language (e.g., Solidity), which is then compiled down to low-level Ethereum Virtual Machine (EVM) bytecode. In order to deploy a contract, an externally owned account publishes a **create** transaction, which includes the contract code as storage. When this transaction is processed, the contract code will be written on the blockchain, and a contract address is generated. If the contract contains a constructor, this function is executed immediately by the miners.

A recently added EVM instruction³ introduced a new way of contract deployment [80]. Previously, a contract address is generated during deployment, and it was not possible to securely predict which address this would be. The new **create2** transaction deterministically calculates the address from the hash of the contract code instead. This method allows users to reference and send coins to contracts that do not exist yet. The secure binding of code and address guarantees that they can always deploy the code later, if necessary.

³This instruction is active since the Constantinople fork of march 2019

Ethereum Gas Model

Transaction fees for Ethereum transactions are paid in *gas*, which is an internal Ethereum currency. Transaction fees in Ethereum are essential to prevent Denial of Service (DoS) attacks, which could force all miners to run unnecessary long code and block their verification resources. Therefore, the amount of gas depends on the size and complexity of transactions. The exact gas value is measured by accumulating the costs of every EVM instruction in the code. All instructions in Ethereum have a fixed amount of gas assigned to it [186]. If a transaction does not contain a sufficient amount of gas, the miners stop the contract execution and revert all changes to the state. Additionally, to the gas amount, every transaction also specifies a gas price, which defines the exchange rate between gas and ether. As miners will always consider transactions with the highest revenue first, the gas price influences how fast a transaction will be processed. Just as in Bitcoin, these transaction fees underly the market demand and rise when blocks get full. In Ethereum, the block size is bounded by how much gas can be used for evaluating all transactions inside it. Table 3.2 shows the relation of gas prices to the confirmation times. For this thesis, we will consider a medium priority of transactions, which means all transaction costs are computed with a gas price of 3 Gwei. We chose this value because it leads to approximately one minute of confirmation time, which still means the protocols will proceed reasonably fast (especially compared to Bitcoin). At the same time, the gas prices are low enough to be comparable with related works. For all calculations in this thesis, we choose an exchange rate of 162.43 euros per Ether, which corresponds to the 180 day average exchange rate calculated on the 24th of February 2020. The data was taken from etherscan.io [78].

Priority	Gas Price	Waiting Time	# of Blocks
low	1 GWei	3640.666667 sec	317.6
medium	3 GWei	326.3333333 sec	27.35
high	6 GWei	37.66666667 sec	2.45
very high	12 GWei	24.66666667 sec	2

Table 3.2: Average time and number of blocks that it takes until an Ethereum transaction is processed.

The price per instruction varies largely, where the overall idea is that instructions that require a lot of the miner’s resources are more expensive. For instance, addition costs 3 gas and multiplication, or modulo operations require 5 gas. Some

cryptographic operations like signing and hashing have their own instructions in the EVM language, i.e., evaluating the **Keccak-256** hash on a 32 byte input takes 36 gas. Storage is especially expensive in Ethereum since every stored value takes space in the state, which all miners have to store in highly accessible memory. Storing a 32 byte word to the Ethereum storage costs 20000 gas and 5000 if already allocated storage is reused. Storing values to volatile memory instead is much cheaper and only requires 3 gas units. A special EVM instruction called **selfdestruct** allows users to deactivate contracts, such that they can be excluded from the miner’s state. This instruction can even lead to the payout of gas to the users.

3.2.2 Designing Secure Contracts

An essential component for the construction of secure smart contracts is the concept of *timeouts*. Whenever an input of a party is required, we construct a timeout around the expected message, which is large enough that an honest party always has sufficient time to react. If an expected message is not received in time, we consider this a faulty behavior. The other contract participant(s) can then trigger a timeout function, which verifies the misbehavior and punishes the party for the missing input. For the secure design of smart contracts, it is crucial that honest parties must never be punished, and therefore, the timeout is sufficiently large. In this thesis, we denote this maximal waiting time as the blockchain delay Δ . We note that in most cases, parties will react reasonably fast, but an honest party’s response could nevertheless be delayed up to Δ rounds (but not longer). Additionally, we require that every transaction and state change is only considered valid, after it has been confirmed by a few more blocks. This security measure mitigates the risks of temporary forks.

When designing secure smart contracts, we always try to identify malicious behavior. Fault attribution is necessary whenever the contract ends up in a state which does not happen during honest behavior. We distinguish between *uniquely* and *non-uniquely attributable faults*. Uniquely attributable faults occur when a malicious party does not follow the protocol, and the other participants can convince the contract about this fact. This case occurs, e.g., when a party signed two contradictory statements, or when a timeout expires. A fault is non-uniquely attributable if some participant of the protocol knows that some other party is dishonest, but they are not able to prove it to the contract. A standard example is a situation when a message is sent directly from a party \mathcal{P} to party \mathcal{Q} . In this

case, \mathcal{Q} might claim that the message was not received while \mathcal{P} claims that he sent it. The parties know who of them is dishonest, but neither can prove it. Uniquely attributable faults are easy to handle since we can instruct the contract to punish the cheating party financially. Non-uniquely attributable faults are harder to deal with as it is not clear who should be punished and which party is telling the truth.

Grieving. Another factor that needs to be considered for a secure and fair contract design is how much fees each party needs to pay. Ideally, the fee burden is equally distributed over all protocol participants, and the maximum amount of fees can be predicted before the protocol starts. This is often not possible when one of the parties misbehaves. In these cases, it would be ideal if the faulty party can be identified and has to carry the fees.

If one of the parties can force another party to pay a much higher share of the fees, we call this a grieving attack. More precisely, a grieving factor of 2 : 1 means that it costs roughly x coins to force another party to pay $2x$ coins in fees. Ideally, the factor is 1 : 1, such that no party has an advantage, or even better, every fault can be attributed, and the misbehaving party carries all fees.

Grieving can also be applied to coin deposits or so-called *collateral* [74]. For the security of some smart contracts, parties need to lock coins for a certain time, e.g., penalty deposits or locking coins for routing payments in payment channels (cf. Section 3.3). For the duration of this locked deposit, the owner cannot use the coins for any other purpose, and the opportunity costs are often regarded as *collateral costs*. Grieving occurs in this case as well, when a malicious party forcibly prolongs the duration of deposits, which increase costs for honest parties. In particular, smart contracts need to ensure that any locked deposit is unlocked eventually, and the lock time is upper bounded.

3.3 Scalability Solutions

In the previous sections, we explained how Bitcoin and Ethereum work and why they have an inherent scalability problem. Recall that their transaction throughput is limited because blocks have only a fixed size and there needs to be sufficient time between blocks such that they can propagate through the underlying gossip network. In this section, we discuss previously proposed scaling solutions and analyze how they can help to reduce transaction costs and confirmation times. Proposals for fixing this scalability issue can be categorized as first or second layer

solutions. Layer one proposals aim to change the consensus rules of the blockchain technology to increase the transaction throughput. Layer two solutions work on unmodified cryptocurrencies and aim to scale through off-chain protocols that only utilize the blockchain for the setup and for settling disputes.

3.3.1 Changing Blockchain Parameters

The bottleneck of the systems is the peer-to-peer gossip network, which limits the speed of block propagation. If blocks get too large or there is not sufficient time between blocks, nodes on the edges of the network will not receive blocks in time to mine competitively. Such a setup would lead to centralization in the long term, which contradicts the goals of the systems. The authors of [58] analyzed the Bitcoin network and concluded that the block size should not be increased to more than 4 MByte, and the blocks should have at least 12 seconds to propagate through the underlying gossip network. In their study, these parameters ensure that 90% of miners would have sufficient bandwidth and connectivity to continue mining. While this measure could help Bitcoin to support 26 Transaction per Second (TPS), it is not enough to reach the throughput of centralized systems like Visa credit card network with 2000 TPS [58].

Some protocols like Fibre [138] or Kadast [168] optimize the message propagation in Bitcoin and others like the bloXroute project [118] propose scaling through a different network architecture. Their idea is to use semi-trusted relays nodes that collect and distribute block and transaction data faster than they would be in gossip networks. If the network information propagation speeds up, it would be possible to increase the block size even further than the bounds found in [58]. Sometimes, these scaling approaches are also called *layer zero* scaling [103].

3.3.2 Scaling the Consensus Layer

Projects which propose changes to the consensus protocols are often classified as *layer one* scaling solutions. They cannot be applied to existing cryptocurrencies easily, as they require an entirely new blockchain protocol which must be accepted by all miners.

A broad range of blockchain consensus protocols has been proposed in the last years, and we refer the reader to [15] for a detailed overview. Here we present a rough categorization and name only a few typical protocols for each category. Traditionally, consensus is a problem from distributed systems or MPC, often with

a fixed network of users (i.e., PBFT [48]). Some consensus protocols approach the problem by electing a leader, i.e., to propose new blocks. The blockchain consensus of Bitcoin [151] and Algorand [88] fall in this category. In hybrid protocols, the network elects one or more committees to speed up the process, e.g., Chainspace [19] or Omniledger [119].

Sharding

A specific form of committee style consensus is sharding. In traditional consensus protocols, the system slows down when more parties join, as either the number of rounds, message, or communication complexity grows in the number of participants. Sharding systems aim for the opposite, the system should get faster if more miners join. The key idea is to divide miners randomly in committees (or shards), where each committee is responsible for a subset of transactions. These shards can then reach consensus on their transactions and publish a (partial) chain faster than the overall network could. At the end of a predefined epoch, the sub-chains of each shard are combined. To prevent that corrupted miners take control over a shard, the assignment of miners to shards is random. But many sharding schemes require a lower adversarial bound than traditional cryptocurrencies. A comparison of sharding schemes [184] found that sharding protocols can lead to a much higher transaction throughput, i.e., Rapidchain [189] can support around 7300 TPS and Omniledger [119] around 10000 TPS when the adversary controls at most 12,5% of the computation power.

Blockchain Data Structure

Some distributed ledger protocols propose to use a Directed Acyclic Graph (DAG) instead of a blockchain for organizing transactions. In Tangle [162], for example, transactions are proposed directly without the need for a central mined blockchain. In DAG protocols, the resulting data structure is not linear, but every transaction (or block) may reference more than one predecessor. The GHOST protocol proposed by [175] proposes such a method for blocks, where uncles, which would be considered invalid forks in Bitcoin-style blockchains, are included in the DAG of blocks. The Ethereum design [186] used a blockchain protocol that follows a GHOST variant. The scalability of these protocols comes from the fact that blocks do not need to propagate through the network entirely, as two or more blocks can (to some degree) be published in parallel and can all be included in the valid DAG. This extension allows Ethereum to have a block creation times of 10 – 20 seconds.

Alternative Mining Puzzles

Yet another approach to scaling blockchain consensus are Proof of Stake (PoS) protocols like Ouroboros [116] and the Ethereum Casper protocol [39]. In PoS schemes, miners do not invest their computational resources but their financial ones. Instead of solving PoWs they lock a certain amount of coins for the purpose of mining, called their *stake*. The protocol randomly selects the miner for the next block proportionally to the size of his stake. As it does not require miners to waste energy on hashing, PoS is considered a more sustainable alternative to PoW blockchains. Additionally, it would lower the costs that miners have to invest, which should lead to lower transaction fees. The scalability comes from the fact that consensus without mining can be reached much faster, and this would reduce block times. Current proposals for secure PoS systems are purely academic and thus, their transaction throughput has not been measured. Nevertheless, the main reason for PoS is to replace the resource waisting PoW schemes and not to improve scalability.

3.3.3 Scaling Through Off-Chain Protocols

Another line of proposals from both academic and industry consider off-chain solutions that work on the so called *second layer*. In contrast to first layer solutions, they aim to increase transaction throughput of existing cryptocurrencies without changing their consensus protocol. They rely on optimistic protocol execution where parties first try to agree without the blockchain and only rely on this expensive and slow component in case someone disagrees. This approach makes off-chain protocols directly compatible to current existing systems. The main idea that these solutions have in common is that a large portion of (transaction) data is processed off-chain between users and are not sent to the blockchain. In contrast to the overall mining network there is no trust assumption on the users, which means they could potentially deviate from the protocol. Therefore second layer solutions use the underlying blockchain (also called parent chain) to secure the user funds during the off-chain phase. While we give a high level overview of existing schemes here, we refer the reader to [103] and [93] for a full overview on existing off-chain proposals.

Payment Channels. The most prominent off-chain solution approach to increase the transaction throughput in blockchain technology is given by payment channels, which allow users to send many transactions off-chain and only commit the final

3 Blockchain Technology

distribution of coins to the parent chain. The first proposal for payment channels was made by Spilman and Hearn [176, 96] who proposed payment channels on Bitcoin as early as 2011 [95]. They construct a unidirectional payment channel using a Bitcoin transaction that can either be spent by the issuer after a timeout t or by two signatures of both the issuer and the recipient of a payment. This technique allowed the issuer to promise increasing shares of the locked money to the recipient (by sending his signature). The recipient would eventually send the latest statement, which represents all accumulated shares to the blockchain and append both his own and the issuers' signature. The timeout t prevents that the issuers' money is locked forever in the channel.

More advanced proposals [61, 161] build bidirectional channels, which allow two users to re-use locked coins off-chain. Bidirectional channels require that channel updates invalidate older channel distributions such that users cannot send outdated states to the parent chain and get more coins than they deserve. In Bitcoin, this is done with refund transactions. Whenever a party proposes a state update, it will ask the channel partner to sign a refund transaction for the previous state, which will make it impossible to collect the coins on-chain from it (in time). In Ethereum, bidirectional channels can be implemented easier by using version numbers or counters [148, 165].

Any channel lifetime can be separated into three distinct phases. During the *opening or funding phase* both players commit their funds on-chain to a multi-signature funding transaction or smart contract. When all funds are locked, the channel is considered to be *open*. If Alice locks c_{Alice} and Bob c_{Bob} , the initial channel balance reflects this distribution. In order to send a payment, the parties update their channel balance and confirm it with their digital signature. For example, if Alice wants to send q coins to Bob, she updates the channel balance to $c_{\text{Alice}} - q$ and $c_{\text{Bob}} + q$. Then she signs the new channel balance and sends it and her signature to Bob. If he also sends his signature to Alice, the update concludes, and both parties have the guarantee that they can enforce the newly updated balance on the blockchain (if they must).

The security that channels offer is that any fund distribution and state that both parties agreed on during the off-chain phase can be enforced on the blockchain within some predefined absolute or relative time period if at least one of the channel participants is online and reactive. If the channel should be closed or if one party notices misbehavior, it sends the latest (signed) channel balance to the blockchain. Then the channel partner will have a predefined time to react and send his own latest version or invalidate any outdated version. After this timeout, the channel

is closed, and the funds are paid out.

Payment channels offer high scalability for many payments between two parties. As long as neither of them aborts from the protocol or closes the channel, they can send transactions off-chain, without additional delays or fees. The parties can only move the money, which is locked on the parent chain, which means they might have to continue on-chain or reopen the channel if a party has insufficient funds for the desired payment.

State Channels. Most literature focuses on payment channels, but some constructions also allow the extension of state channels [148, 71, 56, 86]. They allow a set of parties to execute complex smart contracts off-chain. As long as all parties are honest and agree on the state transitions, the blockchain is contacted only during funding and closing. The update, i.e., the proposal of a new state of a channel contract, is performed off-chain. However, once parties start to disagree, they have to resolve their dispute on-chain and perform the contract evaluation via the blockchain. This is an additional step, which is not necessary in payment channels.

The opening of a state channel works analogous to payment channels. Once the channel is open and funded, the channel state can be updated, which means users can send transactions to each other. This step, often called state transition [93, 56], is executed off-chain. Again, the closing phase requires on-chain interaction by at least one of the players and ensures that the latest update (the last *state*) is enforced on-chain. While in the optimistic case when all parties are honest, state channels are very efficient, a potentially heavy computation might need to be done on-chain in case of disagreement. Just as payment channels, state channels need to provide mechanisms that allow all channel participants to enforce the off-chain state and prevent that money is locked forever.

Additional Works on Payment Channels. Some channel constructions also allow routing of transactions in so-called channel networks [161, 148, 135, 136]. We give a detailed overview of channel network proposals in 4.1.2. Other proposals [148, 68, 56, 35] extend channels to support n parties instead of just two. We will discuss this extension in more detail in Section 4.7.

Payment channels require, that channel participants continuously need to watch the blockchain, in case a malicious user tries to close the channel with an outdated state. In particular, this requires that channel participants are online and have a running node of the underlying blockchain. One factor that influences this problem

is the closing timeout, which says how much time some party has to react to a proposed channel closing. If this parameter is high, e.g., one day, parties only need to check the blockchain for channel-related messages roughly once every 24 hours, but at the same time, this means a channel closing might take one day to finish. A more practical solution for the always online requirement is provided by watchtower services [141, 11, 110]. Users send their latest state to watchtowers before they go off-line and trust the watchtower to complain on their behalf.

Arbitrum. The disadvantage of state channels, i.e., the potentially heavy on-chain execution in case of dispute, is addressed by the Arbitrum proposal [105]. Every smart contract, which Arbitrum models as a Virtual Machine (VM), to be executed off-chain has a set of manager parties responsible for correct VM execution. As long as managers reach consensus on the VM state transitions, execution progresses off-chain in a similar fashion as state channels. But, in case of dispute, managers do not perform the VM state transition on-chain. Instead, one manager can propose the next VM state, which other managers can challenge. If the newly posted state is challenged, the proposer and the challenger run an interactive protocol via the blockchain, so-called *bisection protocol*, in which one disputable computation step is eventually identified and whose correct execution is verified on-chain. Hence, instead of executing the entire state transition on-chain (which might potentially require a lot of time/space), only one computation step of the state transition has to be performed on-chain in addition to the bisection protocol. The Arbitrum protocol works under the assumption that at least one manager of the VM is honest and challenges false states if other managers post them. Since the blockchain interaction during the bisection protocol is rather expensive, Arbitrum uses monetary incentives to motivate managers to behave honestly and follow the protocol.

TrueBit. Another solution that supports off-chain execution of smart contracts using incentive verification is TrueBit [177]. For each off-chain execution, the TrueBit system selects (randomly) one party, called the *solver*, that is responsible for performing the state transition and inform all other parties about the new contract state. The TrueBit system incentivizes parties to become so-called verifiers and check the correctness of the computation performed by the solver. In case they detect misbehavior, they are supposed to challenge the solver on the blockchain and run a *verification game*, which works similarly to the bisection protocol of Arbitrum. Similar to Arbitrum, TrueBit relies on the assumption that there is at

least one honest verifier that correctly performs all the validations and challenges malicious solvers. In contrast to Arbitrum, all inputs and the contract state are inherently public even in the optimistic case when everyone is honest.

Commit Chains. Another off-chain construction relies on a central but untrusted operator, who collects all payments or state updates off-chain and computes a new off-chain state. This is done once every epoch, and then a short commitment to this new state is submitted to the blockchain. These proposals are sometimes called *commit chains* and come in different varieties [111, 160]. Commit chains were designed to reduce the high collateral costs that exists for other off-chain solutions, e.g., for payment channel networks. But an essential distinction between commit chains and channels is the finality of the off-chain transactions. If a user proposes an off-chain state change in a channel and the other channel participants sign it, this change is considered final or valid. This means the user has the guarantee that he can enforce it on-chain. In commit chain scenarios, on the other hand, any proposed change is only final after the operator sends a commitment to the blockchain (at the end of an epoch).

The first proposal for a commit chain was called Plasma, initially introduced by Poon and Buterin [160]. Nowadays, there exists a whole family of plasma protocols⁴ Plasma chains are built on top of the Ethereum blockchain and have their own operator who is responsible for validating off-chain plasma transactions. In regular intervals, he posts a short commitment about the current state of the Plasma chain to a smart contract on the Ethereum blockchain. Additionally, he informs all users about the full, or the relevant parts of the state. The regular commitments are in the form of Merkle tree roots over the whole state of the plasma chain, and provide checkpoints of the Plasma chain to the users. As long as the user can verify his inputs were included in the root, he knows that his inputs were processed. In case the operator cheats, the plasma user can at least enforce the state of the latest checkpoint in the plasma smart contract, by providing that state and a Merkle tree path to the root.

The plasma protocol promises that parties can exit the Plasma chain with all their funds. A recent work analyses the limitations of commit chain proposals [70] and identifies a significant efficiency trade-off of current proposals. If the operator is caught cheating, either a so-called mass exit can occur where all honest parties simultaneously leave the system, or an honest party is forced to send lots of data

⁴An overview of plasma proposals can be found at <https://ethresear.ch/t/plasma-world-map-the-hitchhiker-s-guide-to-the-plasma/4333>.

to the blockchain to prove the latest state is valid. Both scenarios can lead to a high transaction load on the blockchain and to very high fees for honest parties.

While the original goal of Plasma [160] was to support arbitrary complex smart contracts, to the best of our knowledge, there is no formally specified protocol yet that would achieve this goal securely (there currently exist multiple proposals for Minimal Viable Plasma (MVP) [38]). Moreover, the plasma research community currently conjectures that Plasma with general smart contracts might be impossible to construct [16].

In summary, applications that require a fast (reliable) state progression are not ideal candidates for commit chains and should rather rely on state channels instead, as they offer instant finality. Commit chains, on the other hand, allow cheaper off-chain transactions (as no collateral is involved) with delayed finality. The Nocust paper [111] discusses how collateral can help to achieve instant finality over the operator.

3.4 Formal Treatment of the Blockchain

Blockchain protocols are often referred to as consensus protocols. Traditionally, a consensus protocol in cryptography is defined as a means to enable a set of participants to agree on value [84]. In the context of cryptocurrencies, the miners need to (repeatedly) reach consensus on the validity of transactions or state updates (in Ethereum). A particular challenge to blockchain consensus is that the underlying network is permissionless, meaning that peers can join and leave at any time. Additionally, the gossip network (in practice) is unstructured and does not give perfect guarantees on (fast) message transfer. Some previous works [84, 85, 158, 13] have analyzed cryptocurrencies and the guarantees they can provide. We will not present these works in detail, but we summarize the security guarantees and assumptions that we take into consideration in the next sections of this thesis. In Chapters 4 and 5, we work in the UC framework introduced by [44]. At the same time, in Chapter 6, the blockchain is modeled as an oracle, and we rely on its security properties in the standalone model (more details on the distinction of these models can be found in Section 2.2). We give more details on the exact modeling in each chapter.

3.4.1 Security Provided by the Public Ledger

The Bitcoin Backbone protocol [84] analyzes Bitcoin in the standard model under the assumption that more than half of the computing power in the system is controlled by honest miners. The paper and its follow up works [115, 85] identify three important features of blockchains, namely the *common prefix property*, the *chain quality property*, and the *chain growth quality*.

Common Prefix This property guarantees that all honest players agree on a large common chain of blocks, the prefix of the blockchain. While the parties might have a different view on the last k blocks of the chain, the rest of their chains are identical with overwhelming probability. To account for the uncertainty of the last blocks, we assume that honest parties only accept blocks that are confirmed at least k times.

Chain Quality This property argues about the fraction of blocks in the chain that were mined by honest miners. This property guarantees that honest miners will eventually mine blocks and thus control a (potentially small) percentage of blocks in the chain.

Chain Growth This property was originally proposed in [115] and guarantees that after some rounds, the chain gets extended by at least a few blocks. Chain growth guarantees that the protocol cannot be halted, i.e., by DoS attacks, and eventually, new blocks will be found.

With these properties, it can be shown that Bitcoin provides state machine replication with its two properties: liveness and persistence. *Liveness* means that valid transactions from honest parties are guaranteed to be included within the next $\Delta - k$ blocks. By the common prefix property, we know that the transaction will be part of the chain of all other honest parties within k more blocks⁵. This is why we upper bound the time to send a transaction by time Δ ⁶. *Persistence* guarantees that eventually, all users have the same view on the current state of the blockchain (i.e., the processed transactions and their order). In addition, it says that blockchains are *immutable*, which means that once transactions end up in the blockchain (deep enough), they cannot be reverted. Again, the common prefix

⁵For most practical purposes, k is chosen as a small constant, i.e., in Bitcoin, it is generally believed that for $k = 6$, a block can be assumed final.

⁶More precisely, we work in the synchronous communication model and say that it takes at most Δ rounds

property helps to ensure persistence and immutability only after some time has passed, where block confirmations measure time. Formally, a block b_i is confirmed k -times if there exists a valid chain extending b_i with k further blocks. Once block b_i has been sufficiently often confirmed, we can assume that all honest parties agreed on the transactions in block b_i and that they cannot be reverted. In particular, we assume that all honest parties agree on the order of the chain starting from the genesis block b_0 up to block b_i . For a more detailed analysis of Bitcoin and the full security analysis of the above-stated properties, we refer the reader to [85] in the synchronous and semi-synchronous setting, and [158] in the asynchronous setting.

In the work “Bitcoin as a Transaction Ledger: A Composable Treatment” [13], the authors analyze the security of Bitcoin in the UC framework. In particular, they specify Bitcoin as a ledger functionality in the Global UC model of Canetti [45]. The ledger of [13] is a global ideal functionality, which holds the current state of blocks and transactions and provides them to parties. Parties can also send transactions to the ledger, which are included in blocks if they are correct (with respect to the blockchain state and the transaction validation rules).

3.4.2 Communication Model

So far, we have discussed permissionless blockchain protocols in this section. But in the further sections of this thesis, we will analyze another kind of protocol, where we consider interactions of a fixed set of mutually known participants. Therefore, we consider synchronous, round-based communication between parties, where parties are always aware of the current round. Formally, this can be modeled by a global clock [109, 84, 13] for which we omit the detailed modeling here. Whenever parties communicate, we assume that sending a message takes at most one round. If a party (including the adversary) sends a message to another party in round i , then it is received by that party at the beginning of round $i + 1$. Hence, rounds can be understood as a measure of real time⁷. We assume that local computation is instant (or at least negligible compared to the time to send a message) for this model. The adversary can decide about the order in which the messages arrive in a given round. However, we assume that he cannot change the order of messages sent between two honest parties (this can be easily achieved by using, e.g., message counters). Additionally, we assume direct secure channels, which means that messages are authenticated and private. This is a simplifying assumption for our

⁷A round can be translated into a few seconds, which can be viewed as an upper bound on however long it takes to send a message between two honest and active parties

3 Blockchain Technology

protocols, and in practice, additional measures need to be taken, i.e., to encrypt and authenticate messages.

We will abstract from the exact workings of the blockchain and only consider interactions of the parties with an idealized ledger, that fulfills the above-stated properties. Formally we say that the proofs in this thesis are based on the *blockchain assumption*. This means our protocols are secure as long as the above properties hold. To ensure this, we have to consider large enough Δ and k . In order to model liveness, we assume that it takes at most Δ rounds for a party to send a message to the blockchain. This parameter can vary for different blockchain systems but must always be an upper bound on the maximal number of communication rounds that it takes to send a transaction to the miners, get included in a block of the chain, and get confirmed by at least k blocks.

In the formal models of all of the works in this thesis, we also abstract the fees necessary for every transaction. While we do provide an analysis of the costs during the benchmark chapters, we assume that parties will always send enough transaction fees to guarantee that a transaction will be confirmed by time Δ .

4 Virtual Payment Channel Hubs

Summary. This chapter summarizes the PERUN protocol, which was published in “Perun: Virtual Payment Hubs Over Cryptocurrencies” [69]. The applications that motivate this work are scalable micropayments – high-speed and low-cost transfers of tiny amounts of money. As discussed in the previous chapter, current blockchain systems such as Bitcoin (cf., Section 3.1) and Ethereum (cf., Section 3.2) are too expensive and slow to support micropayments. Second layer solutions (cf. Section 3.3) aim to improve the scalability issue by reducing the fees and increasing the transaction throughput. Side- and commit chains are not ideal for micropayments because the delayed finality limits the speed of transaction confirmation. Payment channels, on the other hand, seem to be the perfect solution for micropayments between two parties, as they offer instant confirmation times and eliminate the need for fees per transaction.

A remaining issue of payment channels is that opening and closing still come with fees and delays, and they can only connect two parties at a time. Any new connection requires another channel. Next to the costs and delays that on-chain transactions entail, users must also lock-up additional funds for every new channel – and during the channel lifetime, these funds cannot be used for any other purpose. If many users want to connect to each other, they need to lock a lot of funds in parallel. As a countermeasure to this problem, payment channel networks, like the Lightning network [161], have been proposed. They allow off-chain routing of payments over a path of existing payment channels. However, sending payments through existing networks leads to additional delays, since the payment routing requires the interaction of all path intermediaries. This active involvement adds delays and, most likely, routing fees, and thus hinders the fast execution of microtransactions between two not directly connected users.

In this chapter, we introduce the PERUN network, a new type of channel network, which allows high-speed transactions over a path of channels without additional costs and delays. In this work, we consider a network with a star topology, which supports n users that connect to one intermediary hub, which we will call Ingrid for simplicity. PERUN only requires new users to open a single bidirectional payment

4 Virtual Payment Channel Hubs

channel (on-chain) with Ingrid – we call these channel a *ledger channel*. After this setup, any user in the system may open a new connection off-chain with every other user with Ingrid’s help. As the resulting connection has the same properties as a payment channel, but it is opened and closed entirely off-chain, we call it a *virtual channel*. The users can exchange high-speed micropayments through these virtual channels with direct messages, and thus without additional fees and delays.

To build secure virtual channels, we require more complex ledger channel constructions compared to existing payment channel constructions. In particular, we utilize smart contracts for the on-chain channel setup with the hub. This lets us add additional features, i.e., instead of just distributing funds between Alice to Ingrid, we allow that coins can also be assigned to a different set of public keys, for example, the one of Alice and Bob. Then these coins are locked in a virtual channel between these parties and are unlocked only when both, Alice and Bob, agree on a new distribution (or after a timeout).

In this section, we discuss not only how to design contracts that support virtual payment channels, but also how to use them securely. This involves a intricate protocol in which the intermediary but also the two connecting parties need to open and close new connections carefully. We analyze the security of the resulting PERUN protocol to show that it provides security even when parties behave maliciously or collude. In order to prove the protocol security, we provide accurate modeling of smart contracts and security proof in the UC framework (cf. Section 2.4). To demonstrate the feasibility of the proposed protocol, we also provide a proof-of-concept implementation to estimate costs and produce benchmarks.

We note that the system does not require trust, as we can show that no party can steal funds and owed money will always be paid out. Nevertheless, the parties need Ingrid to interact with them and to lock collateral coins such that their channels will be funded from both sides. Her interaction is needed whenever users want to join and leave the system, and when they want to connect to each other. Additionally, she needs constantly watch the network and blockchain for signs of misbehavior of any of the parties. While we can show that she will always get her locked funds back and no user can cheat Ingrid, she will most likely ask for payment for her service that are proportional to the collateral costs of virtual channels.

We start this section with a high-level overview of a simplified version of our protocol, to describe the key ideas and compare our design to related works (cf. Section 4.1.2). In Section 4.2, we give additional notation and background required for our formal UC modeling. Section 4.3 contains the construction of the PERUN

ideal functionality and Section 4.4 contains the full PERUN protocol. We formally prove UC security in Section 4.5. The implementation details are discussed in Section 4.6 and possible extensions and future work are described in Section 4.7.

4.1 Overview

As discussed in Section 3.3, cryptocurrencies suffer from limited scalability, which leads to high transaction fees when many transactions compete to be included in the next block. Additionally, blockchain technologies can only process transactions with low fees with significant delays, which grow during times of high transaction load. In Section 3.1.3, we presented that it can take hundreds of minutes until Bitcoin transactions with medium fees are accepted in the blockchain. Even in Ethereum, where transactions are usually processed in under a minute (cf. Section 3.2.1), delays of minutes and high fees also prevent some applications.

Even small fees and minor delays are a big issue when we consider *micropayments* – rapid transactions of small amounts, sometimes even fractions of cents. In such high-frequent and fragmented payments, delays cannot be tolerated, and constant (even small) fees would quickly accumulate to large amounts. When users mutually distrust each other, micropayments allow customers to pay in a stream of tiny payments for the ongoing usage of a service or good. They can also be applied in the Internet of Things (IoT) context, where smart devices can pay for their power consumption or network communication. This pay-by-the-minute approach helps to minimize the maintenance costs and sets incentives to build more sustainable devices that minimize resource requirements. Another promising application of micropayments is the digital media consumption in the Internet. The common business models of music, movie or news providers are either subscription payment systems or advertising-supported revenue models. Micropayments could open the path for fairer models where users only pay for every consumed item. Business models that currently are financed by online advertisements or data collection, could switch to tiny user payments or donations. But all of these use-cases are only possible if micropayments are enabled.

We propose a new protocol for instantaneous micropayments in star topology networks with a single hub, over Ethereum. To connect to the network, the users only have to connect to the hub through an on-chain deposit once. Once they established such a connection, they can send instant payments to any user in the network without additional blockchain interaction. Our protocol, called PE-

RUN¹, proposes a more efficient payment channel construction. (cf. Section 3.3). However, payment channels can only provide one-to-one micropayments. While multiple works [161, 148, 135] have analyzed how payment channels can be used to route transactions over multiple connections, routing in these so-called *payment networks* introduces additional delays and fees. The PERUN protocol composes smart contract-based payment channels in a novel way, with a technology we call *virtual channels*. Given existing connections between two parties and an intermediary, we can establish a direct (virtual) link between the two parties off-chain, which allows them to send transactions where the intermediary does not need to get involved in each payment. This protocol significantly reduces latency and costs and can support micropayments. For our protocol, we require smart contracts. Thus PERUN works over any cryptocurrency which allows Turing complete scripts. We demonstrate the feasibility of our proposal by providing a prototype implementation of the channel contracts for *Ethereum* (see Section 4.6).

4.1.1 Intuition and Design Ideas

Let us first informally describe our system in which we present the main contributions in a simplified setting. Here we describe the key design ideas behind ledger and virtual channels on a high level and discuss the security provided by the protocol. We later give a formal definition of the PERUN protocol (cf. Section 4.4).

Ledger channels

We denote payment channels that are built directly on top of the blockchain as *ledger channels*². We use this terminology to differentiate clearly between direct blockchain-based channels and the new virtual channels. A ledger channel allows two parties to instantaneously send payments to each other, once the channel is opened. We denote ledger channels as β and the two channel participants that it connects as the channel *end-parties*. For simplicity we call them Alice (or \mathcal{A}) and Bob (or \mathcal{B}), where Alice deposits $x_{\mathcal{A}}$ coins into the channel and Bob deposits $x_{\mathcal{B}}$ coins into it (for some $x_{\mathcal{A}}, x_{\mathcal{B}} \in \mathbb{R}_{\geq 0}$). Figure 4.1 depicts the basic setup of such a channel.

Ledger payment channels are secured by a *channel smart contract* (or channel contract) on the blockchain. A ledger channel is created through two sequential

¹Perun is the god of thunder and lightning in the Slavic mythology. This choice of a name reflects the fact that one of our main inspirations is the Lightning system [161].

²Ledger channels are essentially identical to payment channels from prior work (see, Section 3.3).

4 Virtual Payment Channel Hubs

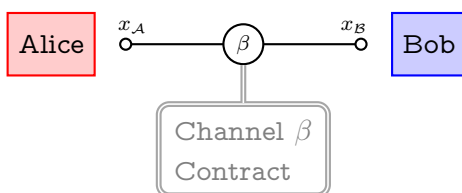


Figure 4.1: Ledger channel β between end-parties \mathcal{A} and \mathcal{B} .

on-chain transactions in an *opening procedure*, initiated by Alice and confirmed by Bob. If Bob does not confirm the opening to the contract, the channel closes, and Alice gets her funds back. Otherwise, the coins of both parties are *locked* in the channel, which means that until the channel β is closed, these coins remain in the channel contract, i.e., the parties cannot use them outside the channel network. The initial *balance* of the channel states that Alice has x_A coins in her *account in* β , Bob has x_B coins in his account, and the overall *value* of the channel is $x_A + x_B$. This balance can be described by a function as $[\mathcal{A} \mapsto x_A, \mathcal{B} \mapsto x_B]$.

Once the opening procedure is finished successfully, and the coins are locked, we consider channel β to be *open*. Now, Alice and Bob can *update* the distribution of the funds in the channel as often as they want off-chain, i.e., without sending transactions to the block. The update mechanism, which consists of two direct messages between the parties, is used for performing payments. If, for example, \mathcal{A} wants to pay some amount $q \leq x_A$ of coins to \mathcal{B} , then the parties perform an update that changes the balance of β , i.e., $[\mathcal{A} \mapsto x_A - q, \mathcal{B} \mapsto x_B + q]$. Naturally, the channel can only be updated as long as both accounts in β have non-negative amounts in them. Performing updates like this guarantees that the total value of the channel never changes. We use counters to keep track of the latest version of the update, and signatures to signal the acceptance of a new state. Both measures ensure that parties only have to store the latest fully signed update. The protocol ensures that they can always enforce it in the underlying contracts.

At any point, the channel β can be *closed* via the contract. Both Alice and Bob can initiate this process by sending a transaction to the contract, which contains the current channel balance $([\mathcal{A} \mapsto x'_A, \mathcal{B} \mapsto x'_B])$. If the other end-party (or counterparty) confirms to the channel contract, that this is indeed the current balance, the channel closes and both parties receive their coins x'_A and x'_B , respectively. In case some party aborts the procedure, the other party can close the channel alone after a sufficient timeout has passed. But in case a malicious party tries to send an outdated state to the contract, the counterparty can (instead of the confirmation) prove to the contract that a more recent valid, signed state exists.

Payment Channel Networks

As payment channels only connect two parties at a time, *payment channel networks* have been proposed [161]. The idea is to use existing channels to route payments off-chain. The setup of a single hop is depicted in Figure 4.2. Instead of a direct channel, we now assume that the parties Alice and Bob are connected via an intermediary party called Ingrid.

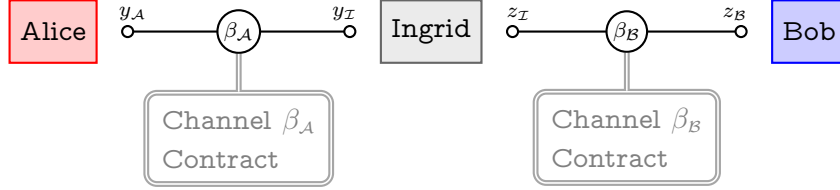


Figure 4.2: Setup for payment routing from Alice to Bob over Ingrid.

The payment routing technique introduced in [161] is a transaction construction called Hashed Time Locked Contract (HTLC). This construction (originally built for Bitcoin scripts) is a conditional payment of coins, where the condition is that the receiver needs to reveal the witness within a certain time limit. This witness r is the preimage of a hash, i.e., the receiver has to reveal r such that $H(r) = h$, and h is fixed in the HTLC. If the expected witness r is not revealed (in time), the coins remain with the sender. This construction can also be used off-chain inside payment channels. Channel participants can update the channel in favor of the receiver if (and only if) a preimage was correctly revealed in time. Therefore, HTLCs can be used to send coins off-chain through existing payment channels (as in Figure 4.2). The key idea is to simultaneously ensure that \mathcal{A} sends q coins to \mathcal{I} , while \mathcal{I} sends q coins to \mathcal{B} . Let us assume that the channel has sufficient funds, i.e., that $y_A \geq q$ and $z_I \geq q$. The payment routing protocol proceeds as follows:

$\mathcal{B} \rightarrow \mathcal{A}$: 1. send hash $h = H(r)$

$\mathcal{A} \rightarrow \mathcal{I}$: 2. HTLC over q coins with witness h and timeout $2t$

$\mathcal{I} \rightarrow \mathcal{B}$: 3. HTLC over q coins with witness h and timeout t

$\mathcal{B} \rightarrow \mathcal{I}$: 4. send witness r

$\mathcal{I} \rightarrow \mathcal{A}$: 5. send witness r

4 Virtual Payment Channel Hubs

As the timeout of the HTLC between Alice and Ingrid is larger than the one between Ingrid and Bob, it ensures that at time t^3 , Ingrid either learned the preimage r from Bob or she knows that the transfer will not happen. In the first case, Ingrid paid for the transaction (either off-chain or because the HTLC was published on-chain), but she also learned the preimage r . This knowledge and the additional time, ensure that the payment will also happen in the channel $\beta_{\mathcal{A}}$. Therefore Ingrid stays *financially neutral* as she receives q coins on one side while she lost q on the other.

A disadvantage of HTLC transaction routing is that the intermediary Ingrid has a lot of control and influence on the transfer. In particular, Ingrid will demand fees for her service every time she is involved in the routing. If the parties Alice and Bob want to send many small microtransactions, they are always dependent on Ingrid's goodwill and cooperation. Additionally, the speed of this exchange is always limited by Ingrid's reaction time. We now show how we can overcome these limitations by using virtual payment channels.

Virtual channels

The main novelty of Perun is the virtual payment channel infrastructure that increases the efficiency of the off-chain payment routing as it does not require interaction with the intermediary Ingrid for payments between Alice and Bob. We apply the same technique that allows parties in ledger channels to update a channel without the blockchain, to create virtual channels that can be updated without the intermediary. While ledger channels are built over smart contracts on-chain, virtual channels are built on top of two ledger channels off-chain. Figure 4.3 illustrates the concept of a virtual channel denoted γ . The channel parties here are \mathcal{A} , \mathcal{B} , and \mathcal{I} , where Alice and Bob are the end-parties of channel γ . γ is built on top of the ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ which we will call *sub-channels* of γ .

Virtual channel opening. \mathcal{A} and \mathcal{B} can establish the virtual channel γ with initial balance $[\mathcal{A} \mapsto x_{\mathcal{A}}, \mathcal{B} \mapsto x_{\mathcal{B}}]$ without blockchain interaction and only with communication with Ingrid. By opening γ , some coins from the parties' accounts in the underlying ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ will be temporarily removed (or locked). More precisely, after opening γ , the balances of $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ change as follows: in channel $\beta_{\mathcal{A}}$ Alice will have $x_{\mathcal{A}}$ coins removed from her account, and Ingrid will

³Where t is sufficiently large such that a transaction can be posted and confirmed on the blockchain (cf. Section 3.4).

4 Virtual Payment Channel Hubs

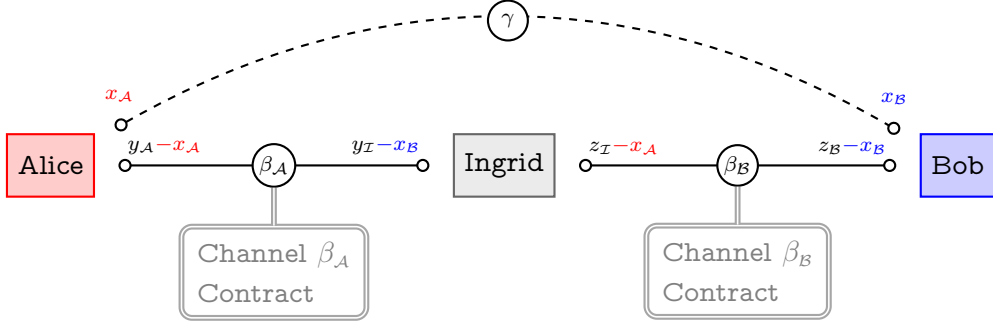


Figure 4.3: Setup for a virtual channel γ between Alice and Bob.

have x_B coins removed. Symmetrically, in channel β_B , Bob locks x_B coins and Ingrid locks x_A coins. The coin distribution after the virtual channel opening is represented in Fig. 4.3.

Opening a virtual channel γ is only possible if all resulting channel balances are non-negative, i.e., $x_A \leq \min(y_A, z_I)$ and $x_B \leq \min(y_I, z_B)$. In other words, \mathcal{A} , \mathcal{B} and \mathcal{I} need to have enough coins in their corresponding ledger channels to open γ . The coins x_A and x_B remain removed from parties' accounts in β_A and β_B for as long as the virtual channel is open. For \mathcal{A} and \mathcal{B} , this situation is similar to locking coins in a newly created ledger channel. Only now they reuse the coins of the ledger channels with \mathcal{I} .

Virtual channel update. Once a virtual channel is opened, it can be updated multiple times, precisely in the same way as the ledger channel, i.e., transferring q coins from \mathcal{A} to \mathcal{B} results into a new balance of γ as before in β . As long as everybody is honest, \mathcal{A} and \mathcal{B} do not need to interact with \mathcal{I} during the update process.

To keep track of the latest update, the end-parties of γ maintain the *version number* $w \in \mathbb{N}$. Initially w is set to 1, and it is incremented after each update of γ . The update procedure is initiated by one of the end-parties. Let us for simplicity assume Alice wants to send q coins to Bob, the opposite case works symmetrically. In this case, \mathcal{A} acts as the *initiator* and \mathcal{B} as the *confirmer*. \mathcal{A} proposes an update of channel γ to a new balance $[\mathcal{A} \mapsto x_A - q, \mathcal{B} \mapsto x_B + q]$ and sends this *update message* $W_{\mathcal{A}} := (m_{\gamma}, \sigma_{\mathcal{A}})$, where m contains the new balances and version

$$m_{\gamma} = \text{update } \gamma \text{ to } [\mathcal{A} \mapsto x_A - q, \mathcal{B} \mapsto x_B + q], \text{ with version number } w$$

and $\sigma_{\mathcal{A}}$ is \mathcal{A} 's signature on m_{γ} . If \mathcal{B} agrees on this update then he replies with

4 Virtual Payment Channel Hubs

$W_B := (m_\gamma, \sigma_B)$ where σ_B is B 's signature on m_γ . At this point the channel is updated to its new balance, and w is incremented by 1.

Recall that transaction routing via HTLCs requires five messages, including communication with the intermediary, who might not have fast reaction times and can force any payment to timeout after a potentially long time t . In virtual channels, on the other hand, the update can be processed as fast as a quick round trip of two messages between two devices. Therefore, virtual channels build the ideal basis for micropayment channels in (hub-based) payment networks.

Virtual channel closing. Each of the channel end-parties $\mathcal{P} \in \{\mathcal{A}, \mathcal{B}\}$ can initiate the channel closing for γ . In order to do so, \mathcal{P} sends the latest update message W_Q that he received from his counterparty \mathcal{Q} (if no update has been performed, then he sends W_Q equal to the initial channel balance with version number 0). When the contract receives W_Q , \mathcal{Q} is notified about \mathcal{P} 's request and replies (within Δ rounds) with the latest update message $W_{\mathcal{P}}$ that he received from \mathcal{P} . Note that if both channel end-parties are honest, they will always agree on the proposed updates, and $W_{\mathcal{P}}$ and $W_{\mathcal{Q}}$ will contain the same message.

When the contract received both messages, it checks which of them $W_{\mathcal{P}}$ and $W_{\mathcal{Q}}$ has a higher version number, and distributes the money according to the balance that is provided in this message. Suppose, for example, the latest state is equal to a transfer of q coins from \mathcal{A} to \mathcal{B} (i.e., $[\mathcal{A} \mapsto x_{\mathcal{A}} - q, \mathcal{B} \mapsto x_{\mathcal{B}} + q]$). Then the channel contract gives $x_{\mathcal{A}} - q$ coins back to Alice and $x_{\mathcal{B}} + q$ coins to Bob in their subchannels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$. Additionally, Ingrid gets $x_{\mathcal{B}} + q$ coins in subchannel $\beta_{\mathcal{B}}$ and $x_{\mathcal{A}} - q$ coins in subchannel $\beta_{\mathcal{A}}$. Overall, closing γ leads to the balance distribution depicted in Figure 4.4.

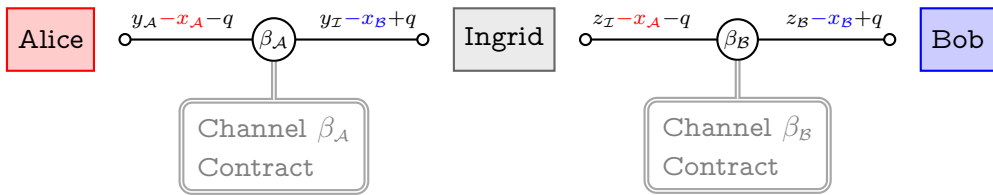


Figure 4.4: Ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ after closing γ .

If we compare the distributions before the opening, and after closing the virtual channel γ , we see that Alice lost q coins while Bob gained q coins. It is important to note that Ingrid did not gain or lose anything since she spent q coins in one channel but, at the same time, received q coins in the other. We say Ingrid stayed

4 Virtual Payment Channel Hubs

financially neutral. This guarantee allows us to update virtual channels without Ingrid’s confirmation, as Alice and Bob cannot change the number of her overall coins. Virtual channels allow her to temporarily use her coins to provide the off-chain infrastructure for transfers between Alice and Bob.

While the distribution of the parties’ balances changes in the ledger channels, their on-chain accounts are not affected (yet). To withdraw the coins from the channel, the parties need to close the underlying ledger channels. However, as long as they do not need the coins for any other purpose, they can also leave the coins in the system and use the channels to send payments to Ingrid or to open new virtual channels off-chain.

Further Challenges and Security Measures

So far, we did not consider that parties act maliciously, e.g., by not reacting (in time) or sending false messages. To prevent that coins are stolen, destroyed, or blocked indefinitely, we need to add additional security measures.

- Channel updates always need to include *digital signatures* by both channel end-parties to show that the update has been approved.
- Channel updates include a *version number* which is increased with every update. They prevent a cheating party from submitting an outdated state. If two different channel updates are provided, the one with the highest version number is considered valid.
- Every message sent to the contract has a timeout for fault attribution, which is large enough to ensure that the messages of honest parties are always accepted. Timeouts guarantee that if an expected message was not received in time, the other channel parties could request an action and potential punishment. For example, during the closing procedure, channel contracts might send funds to the party, which did not misbehave.
- When ledger channels are closed, both parties have a chance to submit their latest channel update. This means closing is not immediate but only after a sufficient waiting period.
- As long as a virtual channel γ is open, our system prevents the subchannels β_A and β_B from being closed. In other words, the parties that opened these ledger channels have to wait with closing them until the financial consequences from the closing of γ are known.

4 Virtual Payment Channel Hubs

- Virtual channels have a special timing parameter called *validity* that \mathcal{A} , \mathcal{B} , and \mathcal{I} agree on during the opening procedure. The validity determines when a virtual channel expires and can be closed by the intermediary. This timeout ensures that (i) Ingrid’s coins cannot be blocked forever and (ii) Alice and Bob have sufficient time to use the virtual channel before the intermediary can close it.

In this section, we omitted some more technical details that the protocol design needs to consider. In particular, we present our scheme in a non-concurrent setting, i.e., we assume that the channels are not opened or updated in parallel and that there is at most one virtual channel built over every ledger channel at any given time.

We formally present the security and efficiency properties of our system in the form of an ideal functionality in Section 4.3 and prove that the PERUN protocol is a UC secure realization of this functionality. We emphasize that our scheme is secure against arbitrary corruptions of \mathcal{A} , \mathcal{I} , and \mathcal{B} , and in particular, no assumption about the honesty of \mathcal{I} are needed.

Consensus on channel opening. A ledger or virtual channel $\delta \in \{\beta, \gamma\}$ can only be opened if all involved parties agree. In particular, Ingrid has to confirm the creation of a virtual channel (and agree on this channel’s validity). Let us emphasize that our protocols guarantee that there is always a consensus among the honest parties, whether a ledger or virtual channel has been successfully opened. This requirement is easily satisfied for the ledger channels (as they are public on the blockchain), but less trivially for virtual channels. The consensus among the honest parties is needed, since a disagreement on the status of γ may lead to misunderstandings. For instance, if Alice thinks that γ has been opened, while Bob believes the opposite, then he will not respond to Alice’s requests to update γ .

Optimistic timings. Opening the ledger channels always takes $\mathcal{O}(\Delta)$ rounds. Opening a virtual channel takes constant time (i.e., time independent of Δ) as it can be processed off-chain. For ledger/virtual channel δ Alice and Bob need to confirm every update. Channel updates always take constant time.

Guaranteed channel closing. Let β be a ledger channel. Both Alice and Bob can request the closing of β at any time (provided there is no virtual channel open over β). Once such a request is made, the channel is closed in time $\mathcal{O}(\Delta)$. Let

4 Virtual Payment Channel Hubs

γ be a virtual channel, and let τ denote its validity. Channel γ is closed in time $\tau + \mathcal{O}(1)$ in the normal case, and $v + \mathcal{O}(\Delta)$ in the pessimistic case.

Guaranteed balance payout for end-users. The end-users of a ledger/virtual channel are guaranteed that the channel’s latest balance is paid out. Concretely, this means for a ledger channel β that coins are transferred back to the accounts of the end-users on the ledger, and for virtual channel γ , it means that the latest balance of the channel is transferred back to the respective ledger channels.

Balance neutrality for intermediary Ingrid. Virtual channels are always *financially neutral* for the intermediary Ingrid. More precisely: suppose γ is a virtual channel built over ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$. Once γ is closed the following holds: if Ingrid loses x coins in $\beta_{\mathcal{A}}$, then she gains x coins in $\beta_{\mathcal{B}}$ (and vice versa).

4.1.2 Related Work

Most related to this protocol are other proposals of payment channel networks. Section 3.3 provides an overview of payment and state channel technologies, so we do not recall all works in detail here. Instead, we compare the works that allow payment routing over non-custodial intermediaries or hubs. In general, these intermediaries are required for the successful protocol execution. However, they are not trusted, i.e., they cannot influence the correctness of the protocol and do not control any of the user funds. It also requires that the intermediaries lock collateral deposits for the time of the transaction routing.

The Lightning Network. The first proposal of off-chain payment routing was made in *The bitcoin lightning network: Scalable off-chain instant payments* [161]. This whitepaper introduced the HTLC technique that allows payment routing in channel networks (cf. Section 4.1.1). When the lightning network was proposed, the Bitcoin currency still lacked a central ingredient to support HTLC. However, the support was added with the *segregated witness* or SegWit Bitcoin improvement (BIP141) [132] in 2017. The underlying problem was *transaction malleability*, which allowed parties to slightly modify transactions before they were included in the blockchain. While this change did not affect the validity of the malleated transaction, it changed its hash and invalidated pre-signed refund transactions, which are crucial for the lightning construction. However, the SegWit Bitcoin fork solved this issue, and the lightning network could be built. Nowadays, the network

connects over 10.000 nodes with more than 35000 channels [1]. In fact, there exist multiple implementations of the lightning protocol [130, 2, 75, 62]. In a recent paper [114], the lightning network was also formalized and analyzed in the UC model.

Sprites [148], the StateChannels project [56], Raiden [165], and Connex [24] also built payment channel networks over Ethereum, which use a similar but simpler routing technique. However, in all of these works, off-chain payment routing still requires interactions with the intermediaries. Only the StateChannels project [56], added a similar technique to virtual channels, which they call meta-channels.

Improvements on Payment Channel Networks

Since the first proposal of payment channels [176] and payment channel networks [161], these technologies were analyzed and extended in various research papers. Two of these works provide an overview of second layer and, in particular, channel technologies [93, 103]. In the following, we present some of the extensions that were made to payment channel networks.

Multi-hop Channel Networks. While we only consider a hub-based payment network, lightning [161] works for open, fully decentralized networks. In particular, payments can be routed over many hops from one point to another. With every new hop, timeouts need to be increased to account for the risk that one link needs to be disputed over on-chain. Therefore, a concurrent work, which is called “Sprites: Payment Channels that Go Faster than Lightning” [148], improved the duration of long HTLC routing, especially with many hops (in Ethereum). They introduce a central registration for the witness, meaning that if one link is disputed over, all intermediaries learn the witness and can immediately close their channels. A recent paper [74] proposed a new technique for Bitcoin-based payment channels, which also only requires a constant collateral lock time.

However, they focus on different aspects of channel networks than we do. Namely, they do not aim to remove the interaction with the intermediaries, but on making the pessimistic time of channel closing constant. Overall, the asymptotic runtime of virtual channels in the PERUN protocol (and its extensions [68, 71]) which influences the time that intermediaries lock collateral, is primarily influenced by the channel validity and not the length of the underlying path.

An interesting attack on multi-hop routing was observed by the authors of [136]. They present the *wormhole attack*, in which two colluding intermediaries on a long

4 Virtual Payment Channel Hubs

payment route eclipse an honest intermediary and steal his routing fees. Instead of closing the HTLC route by revealing the witness, the intermediaries share it with each other directly. As a result, the payment from the sender to the receiver succeeds, but the honest intermediary is not part of it. Thus, the malicious intermediaries collect the routing fees of the honest party, who locks the collateral and behaves honestly but does not get compensated. We note that this cannot happen in the PERUN construction as a virtual channel, once opened is no longer controlled by the intermediaries⁴. The fix for wormhole attacks proposed by [136] is now part of the lightning network construction [103].

Other works focus on efficient routing and pathfinding in large networks. In the lightning network [161], the sender decides which path a payment should take based on his view of the network. Other proposals like SilentWhispers [134] and Flare [163] propose more efficient pathfinding methods through publicly known landmark nodes that hold routing tables. The SpeedyMurmurs [169] protocol proposes an embedding-based approach where every node decides how to route payments further in the directions to the receiver. As routing is straightforward in PERUN we do not discuss these approaches in detail.

With large payment networks another interesting problem occurs. When the channels are regularly used and a high number of payments are routed from one area of the network to another, channels can deplete, i.e., the channel balance is shifted to certain users. If this happens, the channel can only be used to send payments in the opposite direction. The payment channel constructions that we discussed above cannot deal with this situation natively, and the only solution is to add or redistribute funds through an on-chain payment. A solution to this problem is offered by so-called *rebalancing* proposals [112, 74]. On a high level, the idea is to send coins off-chain in a circle, such that the channel funds are distributed evenly after the rebalancing.

Privacy Preserving Payments. Another direction of research papers analyze the privacy aspects of payment channel networks. “Concurrency and privacy with payment-channel networks” [135] proposes privacy-preserving payments which was improved in [136]. Both works guarantee *value privacy*, meaning that no observer can determine how much money was routed over some path. An interesting observation was made in [135], which analyzed that privacy-preserving payment routing is not compatible with *concurrent* payments. In particular, the simultaneous routing of more than one payment over the same link might lead to deadlocks in the

⁴This holds even for multi-hop virtual channel extension as presented in [71].

network. The paper shows that any solution which would solve this issue requires transaction identifiers which in term would lead to linkable payments.

Some works also consider privacy in hub-based networks. The paper “Bolt: Anonymous payment channels for decentralized currencies” [91] constructs a payment channel hub on top of a privacy-preserving currency, i.e., Zcash [190]. The TumbleBit [97] and Trilero [76] protocols provide unlikable payments on top of Bitcoin using a central hub, called a *tumbler*. This untrusted party routes payments but cannot directly link sender and receiver (provided the routed amounts do not reveal this information). The goal is to hide from the hub, who payed whom. While the hub can always see the final balances of all users, he cannot link the single payments. A restriction for both protocols is that the number of coins in each payment is fixed such that the amount does not leak the relation between sender and receiver.

4.2 Preliminaries

In this section, we provide the notation and syntax as well as the formal definitions of the ledger, which we need for the rest of this chapter. We work in the GUC model described in Section 2.4 and rely on the communication model for blockchain communication introduced in Section 3.4.

For the PERUN protocol, we assume a fixed set of parties $P = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ that use the channel network system. We assume that before the protocol starts, a public-key infrastructure setup phase is executed by some trusted party. To simplify the protocol description, we denote the signature of $\mathcal{P} \in P$ on a message m as $\text{Sign}_{\mathcal{P}}(m)$. We say that a tuple $(x_1, \dots, x_n, \sigma)$ is *signed by* \mathcal{P} if σ is a valid signature of \mathcal{P} on (x_1, \dots, x_n) , i.e., $\text{Vrfy}(pk_{\mathcal{P}}, (x_1, \dots, x_n), \sigma) = 1$. We emphasize that the use of a PKI is only an abstraction that helps to describe our protocols. In practice, the trusted setup can easily be realized by posting public keys on the blockchain. To keep the model as simple as possible, we do not include the transaction fees in our modeling.

We use keyword *attributes* attr to refer to certain values in (channel) tuples. Formally, an *attribute tuple* is a function from its set of attributes to $\{0, 1\}^*$. To improve readability, the *value of an attribute* attr *in a tuple* T (i.e., $T(\text{attr})$) is referred to as $T.\text{attr}$.

All messages start with a keyword (e.g., `1c-open`). All communication with the environment \mathcal{Z} and with the ideal functionalities is instantaneous. To account for the blockchain delay, we allow the adversary to delay a reaction of the functionality

4 Virtual Payment Channel Hubs

to a message by at most Δ rounds. We write that some action is executed *within time* Δ . This means that the exact round until when this action is completed is up to the adversary to decide, but Δ is an upper bound.

We introduce two functions π and θ that help to manage the balances of channels.

The balance function $\pi : \{\mathcal{P}, \mathcal{Q}\} \rightarrow \mathbb{R}_{\geq 0}$ describes the current channel balance or coin distribution between two parties \mathcal{P} and \mathcal{Q} . *Adding* $q \in \mathbb{R}_{\geq 0}$ coins to the account of \mathcal{P} in π results in a updated balance function π' which is equal to π for party \mathcal{Q} and for party \mathcal{P} , $\pi'(\mathcal{P}) = \pi(\mathcal{P}) + q$. *Removing* q coins from \mathcal{P} 's account is shorthand for writing $\pi'(\mathcal{P}) = \pi(\mathcal{P}) - q$.

The transfer function $\theta : \{\mathcal{P}, \mathcal{Q}\} \rightarrow \mathbb{R}$ describes the balance change for an update by specifying a redistribution. In particular it specifies how many coins are sent from one party (negative amount) to another (positive amount), hence the following must hold: $\theta(\mathcal{P}) + \theta(\mathcal{Q}) = 0$. *Transferring* q coins from \mathcal{P} to \mathcal{Q} results in a transfer function θ such that $\theta(\mathcal{P}) = -q$ and $\theta(\mathcal{Q}) = q$.

These functions can be added in a natural way, i.e., if f and g are transfer or balance functions for the same parties \mathcal{P} and \mathcal{Q} , then $h = f + g$ is a function $h : \{\mathcal{P}, \mathcal{Q}\} \rightarrow \mathbb{R}_{\geq 0}$ defined as $h(\mathcal{P}) := f(\mathcal{P}) + g(\mathcal{P})$ and $h(\mathcal{Q}) := f(\mathcal{Q}) + g(\mathcal{Q})$.

4.2.1 Channels Syntax

We define a ledger channel over the set of parties P as an attribute tuple β of the form:

$$\beta = (\beta.\text{id}, \beta.\text{Alice}, \beta.\text{Bob}, \beta.\text{cash})$$

and a virtual payment channel γ over P as an attribute tuple of the form:

$$\gamma = (\gamma.\text{id}, \gamma.\text{Alice}, \gamma.\text{Ingrid}, \gamma.\text{Bob}, \gamma.\text{cash}, \gamma.\text{subchan}, \gamma.\text{validity}).$$

Every channel has an identifier $\delta.\text{id} \in \{0, 1\}^*$. The two parties $\delta.\text{Alice}, \delta.\text{Bob} \in P$ are two distinct elements of P called the *end-users* of δ . If $\delta = \gamma$ is a virtual channel, then the party $\gamma.\text{Ingrid}$ is also an element of P (distinct from $\delta.\text{Alice}$ and $\delta.\text{Bob}$) and it is sometimes called the *intermediary*. We define the set of *end-users* of δ as $\delta.\text{end-users} = \{\delta.\text{Alice}, \delta.\text{Bob}\}$ (note that when δ is a virtual channel, then this set does not contain $\delta.\text{Ingrid}$). If δ is a virtual channel then $\delta.\text{all-users}$ denotes the set $\{\delta.\text{Alice}, \delta.\text{Bob}, \delta.\text{Ingrid}\}$, and if δ is a ledger channel then simply $\delta.\text{all-users} = \delta.\text{end-users}$. As it will often simplify the writeup to refer

4 Virtual Payment Channel Hubs

to the opposite end-party (or counterparty) in a channel, we define the shortcut $\delta.\text{other-party} : \delta.\text{end-users} \rightarrow \delta.\text{end-users}$ as $\delta.\text{other-party}(\delta.\text{Alice}) = \delta.\text{Bob}$ and $\delta.\text{other-party}(\delta.\text{Bob}) = \delta.\text{Alice}$, respectively. The attribute $\delta.\text{cash}$ is a balance function for the parties $\delta.\text{end-users}$.

Virtual channels have more attributes than ledger channels. The attribute $\delta.\text{subchan}$ is a function $\text{subchan} : \gamma.\text{end-users} \rightarrow \{0, 1\}^*$ which references the identifiers of the sub-channels $\gamma.\text{subchan}(\gamma.\text{Alice})$ and $\gamma.\text{subchan}(\gamma.\text{Bob})$ over which γ is constructed. Second, the time parameter $\gamma.\text{validity} \in \mathbb{N}$ denotes the channel validity, i.e., the round until which the virtual payment channel stays open.

4.2.2 The Ledger Functionality

We aim to prove the security of the PERUN protocol in the Global UC (GUC) model (cf. Section 2.4) as it allows functionalities to access by functionalities from different sessions. In particular, we model the ledger \mathcal{L} as a global functionality, which makes it available both in the real and ideal world, and moreover, can be used over multiple protocol executions [46, 41].

Functionality \mathcal{L}

Functionality \mathcal{L} runs with a set of parties $P = \{\mathcal{P}_1 \dots, \mathcal{P}_n\}$ and stores a value $p_i \in \mathbb{N}_{\geq 0}$ for every party $\mathcal{P}_i, i \in [n]$ which denotes the number of coins that party $\mathcal{P}_i \in P$ owns. It accepts queries of the following types:

Initialization Upon receiving message $(\text{init}, p_1, \dots, p_n)$ from the Environment \mathcal{Z} (via Sim or \mathcal{A}) such that $p_i \in \mathbb{N}_{\geq 0}$ for all $i \in [n]$, store this tuple.

Add Coins Upon receiving a message $(\text{add}, \mathcal{P}_i, q)$ (for $\mathcal{P}_i \in P$ and $q \in \mathbb{N}_{\geq 0}$) from an ideal functionality \mathcal{F} let $p_i := p_i + q$. We say that the functionality \mathcal{F} added q coins to \mathcal{P}_i 's account in \mathcal{L}

Remove Coins Upon receiving a message $(\text{remove}, \mathcal{P}_i, q)$ (for $\mathcal{P}_i \in P$ and $p_i > q \in \mathbb{N}_{\geq 0}$) from an ideal functionality \mathcal{F} let $p_i := p_i - q$. We say that the functionality \mathcal{F} removed q coins from \mathcal{P}_i 's account in \mathcal{L}

The state of the ledger \mathcal{L} is public, and it maintains a non-negative vector of natural numbers (p_1, \dots, p_n) , where p_i corresponds to the current amount of coins in party \mathcal{P}_i 's account. The parties cannot directly access the ledger. Instead, their accounts are maintained via the smart contract functionality \mathcal{C} (in the real world)

4 Virtual Payment Channel Hubs

or via the ideal functionality $\mathcal{F}_{\text{Channels}}$ (in the ideal world). These functionalities can add or remove coins on a parties' account on the ledger by sending `add` or `remove` messages. We assume that the communication of ideal functionalities and the ledger is instant; this corresponds to the fact that contracts update the state of the ledger immediately after their execution. We model the blockchain delay (cf. Section 3.4) by modeling delays when contract functionalities, i.e., \mathcal{C} are triggered.

We allow the environment \mathcal{Z} (over the simulator Sim in the ideal world and over \mathcal{A} in the real world) to freely remove money from the accounts of corrupted parties. This corresponds to the fact that we are not interested in preventing corrupt parties from acting irrationally and losing money.

4.3 Ideal Functionality

In this section, we state the ideal functionality $\mathcal{F}_{\text{Channels}}$, which defines the behavior of the PERUN channels in the ideal world. This functionality receives messages from and outputs messages to the Environment \mathcal{Z} via the dummy parties Alice \mathcal{A} , Bob \mathcal{B} and Ingrid \mathcal{I} . Recall that dummy parties cannot act on their own. Instead, they forward any message they get from \mathcal{Z} to $\mathcal{F}_{\text{Channels}}$ and vice versa. The functionality also interacts with the ideal world simulator Sim .

Functionality $\mathcal{F}_{\text{Channels}}$
<p>Functionality $\mathcal{F}_{\text{Channels}}$ runs with a set of parties $P = \{\mathcal{A}, \mathcal{B}, \mathcal{I}\}$ and maintains an initially empty channel space Σ. This functionality leaks all messages that it receives to the ideal world simulator Sim.</p>
(A) Opening a ledger channel
<ol style="list-style-type: none"> 1) Upon receiving message $(\text{lc-open}, \beta)$ from party \mathcal{A} in round τ, where β is a ledger channel, s.t. $\mathcal{A} = \beta.\text{Alice}$, (wait at most time Δ) remove $x_{\mathcal{A}} := \beta.\text{cash}(\mathcal{A})$ coins from \mathcal{A}'s account on the ledger \mathcal{L} and go to step 2. 2) Upon receiving message $(\text{lc-open}, \beta)$ from party \mathcal{B} s.t. $\mathcal{B} = \beta.\text{Bob}$ in the next round: <ol style="list-style-type: none"> 2a) If this message was received, (wait at most time Δ to) remove $x_{\mathcal{B}} := \beta.\text{cash}(\mathcal{B})$ coins from \mathcal{B}'s account on the ledger \mathcal{L}, add β to Σ and output (lc-opened) to \mathcal{A}, \mathcal{B}, and Sim. Then accept messages from \mathcal{A} and \mathcal{B} for β as defined in the sub-functionalities (B) - (D) below.

4 Virtual Payment Channel Hubs

- 2b) Otherwise, (wait between $\Delta - 2\Delta$ rounds and) add $x_{\mathcal{A}}$ coins to \mathcal{A} 's account on the ledger \mathcal{L} , output (1c-not-opened) to \mathcal{A} and stop.

(B) Updating a Ledger Channel β or Virtual Channel γ

Accept messages that concern channel δ which is either a ledger channel β or a virtual channel γ stored in Σ .

- 1) Upon receiving a message (update, $\delta.\text{id}, \theta, \alpha$) from a party \mathcal{P} where θ is a transfer function, s.t. $\mathcal{P} \in \delta.\text{end-users}$ and for all $\mathcal{Q} \in \delta.\text{end-users} : \delta.\text{cash}(\mathcal{Q}) + \theta(\mathcal{Q}) \geq 0$. Then, within 3 rounds, send (update-requested, $\beta.\text{id}, \theta, \alpha$) to $\mathcal{P}' := \delta.\text{other-party}(\mathcal{P})$.
- 2a) If in the next round \mathcal{P}' replies with a message (update-ok) replace δ in Σ with a channel $\hat{\delta}$ that is equal to δ , except that $\hat{\delta}.\text{cash} := \delta.\text{cash} + \theta$ and send (updated) to \mathcal{P} .
- 2b) Otherwise do nothing.

(C) Closing Ledger Channel β

Accept messages from end-users of channel $\beta \in \Sigma$.

Upon receiving a message (1c-close, $\beta.\text{id}$) from a party \mathcal{P} s.t. $\mathcal{P} \in \beta.\text{end-users}$ and there is no open virtual channel built over β , do the following (within time 3Δ ^a):

- Add $\beta.\text{cash}(\beta.\text{Alice})$ coins to $\beta.\text{Alice}$'s account on \mathcal{L} .
- Add $\beta.\text{cash}(\beta.\text{Bob})$ coins to $\beta.\text{Bob}$'s account on \mathcal{L} .
- Erase β from Σ .
- Send (1c-closed) to the parties in $\beta.\text{end-users}$ and to *Sim*.

(D)-(E) Opening and closing a virtual channel γ

- 1a) Upon receiving the message $m = (\text{vc-open}, \gamma)$ from parties \mathcal{A} , \mathcal{B} , and \mathcal{I} within two rounds, where γ is a virtual channel and $\gamma.\text{Alice} = \mathcal{A}$, $\gamma.\text{Bob} = \mathcal{B}$ and $\gamma.\text{Ingrid} = \mathcal{I}$, (wait at most time Δ to)
 - remove $\gamma.\text{cash}(\mathcal{A})$ coins from \mathcal{A} 's account and $\gamma.\text{cash}(\mathcal{B})$ coins from \mathcal{I} 's account in $\Sigma(\gamma.\text{subchan}(\mathcal{A}))$.
 - Remove $\gamma.\text{cash}(\mathcal{B})$ coins from \mathcal{B} 's account and $\gamma.\text{cash}(\mathcal{A})$ coins from \mathcal{I} 's account in $\Sigma(\gamma.\text{subchan}(\mathcal{B}))$.

4 Virtual Payment Channel Hubs

- Add γ to Σ ,
 - Output (**vc-opened**) to \mathcal{A}, \mathcal{B} and \mathcal{I} .
- 1b) If within 2 rounds (from receiving m for the first time) you do not receive m from *all* the parties \mathcal{A}, \mathcal{B} and $\mathcal{I} \in \gamma.\text{all-users}$ then (wait between $\Delta - 2\Delta$ rounds to) output **vc-not-opened** to them and stop.
- 2) Wait until round $\gamma.\text{validity}$ where $\hat{\gamma} := \Sigma(\gamma.\text{id})$ is the current version of γ . Then execute the following operations within round $\gamma.\text{validity} + 7\Delta + 5^b$:
- Add $\hat{\gamma}.\text{cash}(\mathcal{A})$ coins to \mathcal{A} 's account and $\hat{\gamma}.\text{cash}(\mathcal{B})$ coins to \mathcal{I} 's account in $\Sigma(\gamma.\text{subchan}(\mathcal{A}))$.
 - Add $\hat{\gamma}.\text{cash}(\mathcal{B})$ coins to \mathcal{B} 's account and $\hat{\gamma}.\text{cash}(\mathcal{A})$ coins to \mathcal{I} 's account in $\Sigma(\gamma.\text{subchan}(\mathcal{B}))$.
- Output (**vc-closed**) to \mathcal{A}, \mathcal{B} and \mathcal{I} and erase $\hat{\gamma}$ from Σ .

^athis is reduced to 2Δ in the optimistic case, i.e., when both $\beta.\text{end-users}$ are honest

^bthis is reduced to $\gamma.\text{validity} + 5$ in the optimistic case, i.e., when all $\gamma.\text{end-users}$ are honest

The functionality $\mathcal{F}_{\text{Channels}}$ maintains a *channel space* Σ – an initially empty set that stores all open ledger and virtual channel tuples. We require that channels have unique identifiers, i.e., for every $id \in \{0, 1\}^*$ there exists at most one $\delta \in \Sigma$ s.t. $\delta.\text{id} = id$. This allows us to refer to channels by their id: $\delta = \Sigma(id)$. For virtual channels $\gamma \in \Sigma$, we additionally require that Σ also contains the two ledger channels $\beta_A, \beta_B \in \Sigma$ which were used to construct γ , i.e., that $\gamma.\text{subchan}(\gamma.\text{Alice}) = \beta_A.\text{id}$ and $\gamma.\text{subchan}(\gamma.\text{Bob}) = \beta_B.\text{id}$.

The $\mathcal{F}_{\text{Channels}}$ functionality is triggered by messages from the parties (messages concerning the ledger channels start with **lc**, and those concerning the virtual ones start with **vc**). A ledger channel β between **Alice** and **Bob** is opened by a message (**lc-open**, β) from **Alice** and a confirmation (**lc-open**, β) from **Bob**. The functionality removes **Alice**'s coins from \mathcal{L} and refunds them if **Bob** does not confirm. Otherwise, the channel is considered open and added to the channel space Σ . A virtual channel γ between **Alice** and **Bob** over **Ingrid** is opened by a messages (**vc-open**, γ) from each party and closed automatically when time $\gamma.\text{validity}$ comes. Ledger and Virtual channels are updated via a message (**update**, id, θ, α), where id refers to the channel that shall be updated according to the transfer function θ . The *update annotation* $\alpha \in \{0, 1\}^*$ is used to guarantee that the parties agree

4 Virtual Payment Channel Hubs

on why a given update happen (see also Eq. (4.1) on p. 82). Channel updates are triggered by one of the channel end users (message `update-requested`) and confirmed by the other (message `update-ok`). Ledger channels are closed with a message (`lc-close, id`).

Note that the parties can play different roles in the virtual channels, e.g., it may happen that virtual channels γ and γ' are open over β , and β .Alice plays the roles of γ .Alice and γ' .Ingrid while β .Bob plays the roles of γ .Ingrid and γ' .Alice, say. We emphasize that the description of the ideal functionality is significantly simplified due to the restrictions on the environment that we make below. Note that (unlike the simplified protocol in Section 4.1.1), our functionality is fully concurrent, and in particular several channel updates can be performed simultaneously, and multiple virtual channels can be open over the same ledger channel β .

4.3.1 Restrictions to the Environment

Below we list the restrictions on the environment that we make to simplify the protocol. Most of them are very natural and ensure that the environment never asks the honest users to do something obviously wrong, e.g., open two different channels with the same identifier, or open a channel without having sufficient funds. These restrictions could be eliminated at the cost of a more complex protocol description.

- The environment never asks the parties to open a channel δ such that δ .id already exists, or when the parties do not have enough funds.
- If the environment asks the parties to open a virtual channel γ then the channel with identifiers specified in γ .subchan exists in Σ , and no closing procedure for them has been initiated.
- The environment never asks to close a ledger channel in time earlier than γ .validity + 7Δ + 5 where γ is a virtual channel whose opening has been initiated by the environment (even if this opening was unsuccessful).
- If the environment asks one of the parties $P \in \delta$.all-users to open a channel δ , then it asks all the other parties in δ .all-users to do the same (in the same round).
- The environment does not perform (or confirm) any update procedures for channels whose closing has been initiated.

4 Virtual Payment Channel Hubs

- If a previous update of a channel δ failed, then the environment will not request a new update of δ .
- The environment always confirms an update that it initiated, and never confirms an update which she did not initiate.
- The environment always instructs corrupted parties (via the adversary) to initiate coin refunds. This restriction allows us to abstract from the explicit refund and lets functionalities always payout coins within one blockchain round (Δ). We discuss in Section 4.5 how this assumption and simplification could be easily removed by adding the influence for *Sim* to instruct \mathcal{F} to not add coins back (e.g., during the ledger channel opening).

A consequence of these restrictions is that in our protocol, we can assume that all the honest parties have the same view on what channels should be open. For example: β .Alice knows that if she received a $(\text{vc-open}, \beta)$ message from the environment, then β .Bob also received such a message (in the same round). This, in particular, means that if β .Bob refuses to participate in the procedure of opening channel β , then he must be corrupt.

4.3.2 Perun Properties

Next, we will discuss how the ideal functionality $\mathcal{F}_{\text{Channels}}$ satisfies the security requirements defined in Sec. 4.1.1.

Consensus on channel opening and on channel update: The ideal functionality $\mathcal{F}_{\text{Channels}}$ always guarantees that honest parties always agree on whether a channel has been created or updated. This is achieved by the notification that the functionality sends to parties. The `lc-opened`/`lc-not-opened` or `vc-opened`/`vc-not-opened` messages ensure that all parties know whether a channel has successfully been created or not (after at most $\mathcal{O}(\Delta)$ rounds). Similarly, the functionality (instantly) informs parties about update requests and completed updates, thus ensuring consensus on updates.

Guaranteed channel closing: A ledger channel β can be closed by any of the parties $P \in \beta$.end-users as long as there does not exist a virtual channel that uses β as a subchannel. In this case, the closing is completed within time at most 3Δ .

4 Virtual Payment Channel Hubs

If an open virtual channel γ uses β as a sub-channel, the parties in β .end-users have to wait until γ is closed, before they can close β . Virtual channels are closed automatically (cf., Step B.2) after the validity is over and the closing procedure takes at most until γ .validity + $7\Delta + 5$. If all participants of the virtual channel are honest, this process will be completed within γ .validity + 5 rounds.

From these two cases, it follows that virtual channels can be closed within a predefined time, depending on validity, and ledger channels can be closed in a fixed time after potential virtual channels are closed. From this reasoning, it gets evident why we need virtual channel validity to guarantee this property.

Guaranteed balance payout for end-users: When a virtual channel is opened, the coins for this channel are taken out of the underlying ledger channels by the functionality $\mathcal{F}_{\text{Channels}}$. The exact number of coins are added back to the sub-channels when the virtual channel is closed, thereby enforcing the coin distribution from the latest virtual channel update (cf., Step B.2). Only after all virtual channels are closed, the underlying ledger channels can be closed (procedure (D)). The functionality *pays out* the latest channel balance, meaning that the coins from the current balance are added to the user accounts in the ledger. These steps guarantee that no coins are created or lost by opening or closing any channels and that the coin distribution during closing is enforced.

Balance neutrality for intermediary Ingrid: To understand how we achieve balance neutrality for Ingrid, we analyze the opening and closing procedures of a virtual channel γ over its two subchannels $\beta_A := \Sigma(\gamma$.subchan(γ .Alice)) and $\beta_B := \Sigma(\gamma$.subchan(γ .Bob)). When the virtual channel is opened γ .Ingrid had a total of $x_A + x_B$ coins removed from her accounts in the subchannels, i.e., $x_A := \gamma$.cash(γ .Bob) from her account in β_A , and $x_B := \gamma$.cash(γ .Alice) from her account in β_B (cf., Step B.1a).

During the closing procedure (Step B.2) she gets $x'_A + x'_B$ coins back to her accounts, i.e., $x'_A := \widehat{\gamma}$.cash(γ .Bob) to in β_A , and $x'_B := \widehat{\gamma}$.cash(γ .Alice) coins in β_B . It remains to show that $x_A + x_B = x'_A + x'_B$. This is guaranteed by the channel update process since it does not allow overall value changes in the virtual channel γ . θ in the update requests has to be a transfer function, which guarantees that for every update $x_A + x_B = x'_A + x'_B$. Hence the balance neutrality holds.

4.3.3 Formal Security Statement

Let κ denote the security parameter (which is given as input to the environment and to the parties). First, $REAL_{\Pi_{\text{channel}}, \mathcal{C}, \mathcal{L}}^{\mathcal{Z}, \mathcal{A}}(\kappa)$ is the output of \mathcal{Z} running the real world protocol Π_{channel} in the \mathcal{C} -hybrid world with adversary \mathcal{A} . Second, $IDEAL_{\mathcal{F}_{\text{Channels}}, \mathcal{L}}^{\mathcal{Z}, Sim}(\kappa)$ denotes the output of \mathcal{Z} running in the ideal world with the $\mathcal{F}_{\text{Channels}}$ ideal functionality and the simulator Sim . In both cases to simplify exposition, we will assume that \mathcal{Z} is from a class of restricted environments, i.e., we will make some explicit assumptions about \mathcal{Z} 's behavior (cf., Section 4.3.1).

We say that protocol Π_{channel} running in the \mathcal{C} -hybrid world emulates an ideal functionality $\mathcal{F}_{\text{Channels}}$ with respect to a global ledger \mathcal{L} and with blockchain delay of Δ rounds, if for any PPT adversary \mathcal{A} there exists a simulator Sim such that for all restricted environments \mathcal{Z} (see Section 4.3.1), we have:

$$HYBRID_{\Pi_{\text{channel}}, \mathcal{C}, \mathcal{L}}^{\mathcal{Z}, \mathcal{A}}(\kappa) \approx IDEAL_{\mathcal{F}_{\text{Channels}}, \mathcal{L}}^{\mathcal{Z}, Sim}(\kappa).$$

We can now state our main security theorem formally.

Theorem 1. *Assume the underlying signature scheme is existentially unforgeable against adaptive chosen-message attacks. Then the protocol Π_{channel} running in the \mathcal{C} -hybrid world GUC emulates an ideal functionality $\mathcal{F}_{\text{Channels}}$ with respect to a global ledger \mathcal{L} and with blockchain delay Δ .*

4.4 The Perun Protocol

In this section, we provide a formal description of the PERUN protocol Π_{channel} . Our protocol Π_{channel} includes interaction with a smart contract, which we must formally define in the UC framework. Therefore Π_{channel} is defined in a \mathcal{C} -hybrid world, where \mathcal{C} is the *contract functionality* that maintains the set of active *contract instances*. We define \mathcal{C} formally in Section 4.4.1. We assume the existence of a public-key infrastructure (cf. Section 4.2).

Challenges from Concurrency. In contrast to the informal description in Section 4.1, we consider a fully concurrent execution here, which means that many channels can be opened, updated, and closed in parallel. In particular, any ledger or virtual channel δ could receive two different updates from two parties in the same round. To ensure the security of the protocol, we must prevent that the parties agree on two different updates with the same version number w in this

4 Virtual Payment Channel Hubs

case. To avoid this scenario altogether and keep the protocol simple, we prevent that δ .Alice and δ .Bob can propose updates in the same rounds. As the update process takes two rounds and the parties should only be able to propose updates in alternating rounds, they can only initiate updates every fourth round. We say δ .Alice can propose updates if the round number $\tau = 0 \pmod{4}$ and δ .Bob if the round number $\tau = 2 \pmod{4}$. Note, that since we assumed that the adversary cannot reorder messages sent from \mathcal{P} to \mathcal{P}' (see Sect. 3.4.2), the version number w will remain synchronized between the parties.

Another potential problem could come from the fact that two update requests that are sent in the same round by \mathcal{P} arrive at \mathcal{P}' in reversed order. Note that this would lead to inconsistent views of \mathcal{P} and \mathcal{P}' on the transfer functions θ in both of these requests. To avoid this issue, we assume that the adversary cannot reorder messages sent between two parties in the same round.

Additionally, we need to address the challenge that several virtual channels are simultaneously opened over the same ledger channel $\beta_{\mathcal{P}}$. Multiple virtual channels imply that – once cheating behavior is detected – honest parties can be forced to wait until the timeout of each open virtual channel has passed before they can close the underlying ledger channel $\beta_{\mathcal{P}}$ that connects them with the malicious party. Recall, that this waiting period guarantees that intermediaries cannot be cheated for their locked coins. When closing the ledger channel $\beta_{\mathcal{P}}$, all final balances of virtual channels must be known. Therefore, we do *not* instruct Ingrid to close $\beta_{\mathcal{P}}$ while there are still open virtual channels. Instead, we let the contract instance $\mathcal{C}(\beta_{\mathcal{P}}.\text{id})$ (that corresponds to $\beta_{\mathcal{P}}$) simply record the information about the outcome x of each virtual. Observe, that there may be multiple such x 's that need to be stored in $\mathcal{C}(\beta_{\mathcal{P}}.\text{id})$ during the lifetime of $\beta_{\mathcal{P}}$ (each x coming from closing a different virtual channel that is constructed over $\beta_{\mathcal{P}}$). To save space in the contract's storage, we simply accumulate all of the values by summing them up. Technically, this is done by defining a transfer function $\text{transfer} : \beta_{\mathcal{P}}.\text{end-users} \rightarrow \mathbb{R}$ that is initially equal to 0 on both inputs. This function keeps track of the number of coins that needs to be transferred between the parties. That is, each time x coins are transferred from $\beta_{\mathcal{P}}.$ Alice to $\beta_{\mathcal{P}}.$ Bob; the function is updated by letting $\text{transfer}(\beta_{\mathcal{P}}.\text{Alice}) := \text{transfer}(\beta_{\mathcal{P}}.\text{Alice}) - x$ and $\text{transfer}(\beta_{\mathcal{P}}.\text{Bob}) := \text{transfer}(\beta_{\mathcal{P}}.\text{Bob}) + x$.

This function is kept in the contract's storage until the channel is closed. During the channel closing, it will be used to correct the amounts of coins that the parties receive. Suppose, for example, that the last balance of $\beta_{\mathcal{P}}$ on which that parties exchanged the signatures is $[\beta_{\mathcal{P}}.\text{Alice} \mapsto y_A, \beta_{\mathcal{P}}.\text{Bob} \mapsto y_B]$. Then as a result of

closing the channel $\beta_{\mathcal{P}}.$ Alice will get $y_{\mathcal{A}} + \text{transfer}(\beta_{\mathcal{P}}.\text{Alice})$ coins, and $\beta_{\mathcal{P}}.$ Bob will get $y_{\mathcal{B}} + \text{transfer}(\beta_{\mathcal{P}}.\text{Bob})$ coins.

4.4.1 The Channel Smart Contract \mathcal{C}

Let us start by constructing the ledger channel smart contract. In our protocol, this contract is modeled as an ideal functionality that is used by the parties in \mathcal{P} . In the UC framework, this ideal functionality is also called a *hybrid functionality*.

As discussed in Section 3.2, Ethereum smart contracts always need to be triggered by transactions in order to execute a function, receive or payout coins or store data. This means we need to anticipate this contract behavior in our protocol. As a contract can also not send messages to parties directly to inform them of a particular behavior, we require honest parties to always watch the contract on the blockchain to see if it changed state after some function call.

Whenever we construct timeouts, we assume that an honest party will trigger the smart contract to ensure that the timeout is enforced. If a message is expected from a party, the functionality will inform the other channel participants once the message arrived. If the message did not arrive in time, other honest parties send a timeout message, that wakes up the functionality, which can verify that the timeout for the expected message expired. In most cases, the functionality will then be able to attribute the fault and punish the malicious party by assigning all funds that are concerned with this fault to the other channel party. Fault attribution and timeouts are presented in more detail in Section 3.2. In the functionality \mathcal{C} , we handle fault attribution in virtual channels in the subroutine (C). Otherwise, the contract consists of the following parts: (A) the subroutine used for constructing a given contract instance and (B) the main execution functionality of the contract.

The *contract functionality* \mathcal{C} maintains the set of active *contract instances*. Each contract instance has a unique *identifier*. We refer to a contract instance with identifier id as $\mathcal{C}(id)$. In our case, each contract instance corresponds to one ledger channel, and, for simplicity, has the same identifier. In other words, a contract instance $\mathcal{C}(\beta.id)$ corresponds to a ledger channel β . When a channel is closed, then the corresponding contract instance terminates (i.e., it is removed from the set of contract instances of \mathcal{C}). Contract instances can be easily implemented as a separate contracts or as a singleton contract storing \mathcal{C} on the Ethereum ledger. For simplicity we consider a separate contract for each instance. A new contract instance $\mathcal{C}(\beta.id)$ is created when \mathcal{C} receives a *constructor message*. We also say that a message m is sent to $\mathcal{C}(id)$ or sent by $\mathcal{C}(id)$ to denote interaction with

4 Virtual Payment Channel Hubs

this specific contract instance. One can also think about it in the following way: every message (other than the constructor message) that is sent to \mathcal{C} contains the identifier id that specifies to which particular contract instance it is addressed, and a similar rule applies to messages sent by \mathcal{C} .

The contract \mathcal{C} defines a transfer function $transfers : \beta.\text{end-users} \rightarrow \mathbb{R}$ initially equal to 0 on both inputs. This function keeps track of the sum of the transfers between $\beta.\text{Alice}$ and $\beta.\text{Bob}$ that were communicated to the contract. In our case, these transfers will come only from the closing of virtual channels. The contract also stores information about virtual channels (built on top of β) that were closed via the contract. Technically, we say that some channel γ is *marked as closed* if it is added to the list of such closed channels.

When both channel end-users are honest, the contract is executed only for the optimistic closing of the channel through a `lc-close` message to the contract (cf. Subroutine (B) Step 4 and 4a). In particular, this means both parties only try to close the ledger channel after all virtual channels are closed. However, if at least one of the channel end parties is dishonest, the other party may initiate the closing of a virtual channel via the contract functionality. In this case, an honest party must be able to prove that the counter party agreed to open a virtual channel. For this purpose both end-users sign *opening certificates* (γ, σ) every time they open a new virtual channel. This statement can later be sent to the contract to prove this fact. Similarly, parties exchange signed closing certificates when they agree to close a virtual channel. An opening certificate $oc_{\mathcal{P}}$ (resp. closing certificate $cc_{\mathcal{P}}$) for party P and channel γ look as following:

$$\begin{aligned} oc_{\mathcal{P}} &:= (\text{open } \gamma \text{ with initial balance } [\mathcal{A} \mapsto x_{\mathcal{A}}; \mathcal{B} \mapsto x_{\mathcal{B}}] \text{ and validity } v) \\ cc_{\mathcal{P}} &:= (\text{close } \gamma \text{ with final balance } [\mathcal{A} \mapsto x_{\mathcal{A}}; \mathcal{B} \mapsto x_{\mathcal{B}}] \text{ and validity } v), \end{aligned}$$

Where each such statement is accompanied by a signature $\sigma_{\mathcal{P}}$ over the respective statement by party \mathcal{P} .

Additionally, the contract must be able to compare channel versions for both ledger and virtual channels. To this end, we use the following terminology to describe channel version tuples: Let $w \in \mathbb{N}$ be a natural number called a *version number*, and $\alpha \in \{0, 1\}^*$ be an update annotation (see Sect. 4.3). Then $(\hat{\delta}, w, \alpha)$ is called a *version of δ* if $\hat{\delta}$ is equal to δ on all attributes except of $\delta.\text{cash}$, and the sum of the balances (the value) of $\hat{\delta}$ is equal to the value of δ . Moreover, $(\hat{\delta}, w, \alpha, \sigma)$ is called a *signed version of δ* if σ is a valid signature of \mathcal{P} on $(\hat{\delta}, w, \alpha)$. If $w = 0$ then we call $(\hat{\delta}, w, \alpha)$ the *initial version of δ* , and we do not require a signature, i.e., we allow $\sigma = \perp$.

4 Virtual Payment Channel Hubs

We define the *winner selection procedure* Win to determine which version of a channel is newer. It was already implicitly defined for V 's and W 's in Section 4.3. Formally it is defined as follows. Let δ be a ledger or virtual channel. Win takes as input a pair $((\delta^0, w^0, \alpha^0, \sigma^0), (\delta^1, w^1, \alpha^1, \sigma^1))$ of signed versions of δ , and returns as output a cash function $\theta : \delta.\text{end-users} \rightarrow \mathbb{R}_{\geq 0}$ defined as follows: let i be such that $w^i > w^{1-i}$ (if no such i exists then choose $i := 0$) and then let $\theta := \delta^i.\text{cash}$.

We are now ready to define the hybrid contract functionality \mathcal{C} formally:

Hybrid Functionality \mathcal{C} .
This functionality captures the behavior of a channel contract instance β .
(A) The contract for channel β opening
<ol style="list-style-type: none"> 1) Upon receiving message $(\text{lc-open}, \beta)$ from party \mathcal{A} where β is a ledger channel s.t. $\mathcal{A} = \beta.\text{Alice}$, remove $\beta.\text{cash}(\mathcal{A})$ coins from \mathcal{A}'s account on the ledger \mathcal{L} and send message $(\text{lc-opening}, \beta)$ to $\beta.\text{Bob}$ and go to step 2) 2) Wait at most Δ rounds to receive a message $(\text{lc-open}, \beta)$ from party $\mathcal{B} = \beta.\text{Bob}$. <ol style="list-style-type: none"> 2a) If this message was received, then remove $\beta.\text{cash}(\mathcal{B})$ coins from \mathcal{B}'s account on the ledger \mathcal{L}. Let $\text{transfers} : \beta.\text{end-users} \rightarrow \mathbb{R}$ be a transfer function for β which initially outputs 0 on both inputs; Send a message (lc-opened) to $\beta.\text{end-users}$ and run subprocedure (B) below. 2b) Otherwise, if no message from \mathcal{B} was recorded add $x_{\mathcal{A}}$ coins back to \mathcal{A}'s account in \mathcal{L} and send message (lc-not-opened) to \mathcal{A}.
(B) The contract $\mathcal{C}(id)$ execution
<p><i>Assumption:</i> for every channel $\delta \in \{\beta, \gamma\}$ each party \mathcal{P} can send at most one message of a given type that concerns δ.</p> <ol style="list-style-type: none"> 1. Upon receiving $(\text{vc-close-init}, \gamma, oc_{\mathcal{P}})$ from $\gamma.\text{Ingrid} \in \beta.\text{end-users}$ in time at least $\gamma.\text{validity} + 2$ (where $oc_{\mathcal{P}}$ is an opening certificate of $\mathcal{P} := \beta.\text{other-party}(\gamma.\text{Ingrid})$ on γ) and γ has not been marked as closed: then send a message $(\text{vc-close-init}, \gamma.\text{id})$ to \mathcal{P} and wait for one of the following messages: <ol style="list-style-type: none"> a) Upon receiving $(\text{vc-already-closed}, \gamma, cc_{\mathcal{P}})$ from \mathcal{P}, where $cc_{\mathcal{P}}$ is a closing certificate of $\gamma.\text{other-party}(\mathcal{P})$ on γ: then mark γ as closed.

- b) Upon receiving $m := (\text{vc-close}, \gamma, W, \text{Sign}_{\mathcal{P}}(W))$ from \mathcal{P} (where W is a version of γ signed by $\gamma.\text{other-party}(\mathcal{P})$): then send m to $\gamma.\text{Ingrid}$.
 - c) Upon receiving $(\text{vc-close-timeout}, \gamma)$ from $\gamma.\text{Ingrid}$ in time at least Δ after you sent the message $(\text{vc-close-init}, \gamma.\text{id})$: then go to subroutine (C) below.
2. Upon receiving message $m := (\text{vc-close-final}, \gamma, oc_{\mathcal{P}}, (V_{\gamma.\text{Bob}}, S_{\gamma.\text{Alice}}), (V_{\gamma.\text{Alice}}, S_{\gamma.\text{Bob}}))$ from $\gamma.\text{Ingrid}$ where $oc_{\mathcal{P}}$ is an opening certificate of $\mathcal{P} := \beta.\text{other-party}(\gamma.\text{Ingrid})$ on γ , each $V_{\mathcal{P}}$ is a version of γ signed by $\gamma.\text{other-party}(\mathcal{P})$, and $S_{\mathcal{P}}$ is a signature of \mathcal{P} on W (or is equal to \perp if W is the initial version of γ), and γ has not been marked as closed: then send message m to \mathcal{P} and wait for one of the following messages:
 - a) Upon receiving $(\text{vc-already-closed}, \gamma, cc_{\mathcal{P}})$ from \mathcal{P} , where $cc_{\mathcal{P}}$ is a signed closing certificate on γ from $\gamma.\text{Ingrid}$ do nothing.
 - b) Else upon receiving $(\text{vc-close-timeout}, \gamma)$ from $\gamma.\text{Ingrid}$ in time $\Delta + 1$ after you sent m to \mathcal{P} : then go to subroutine (C) below.
 3. Upon receiving $(\text{vc-close-timeout}, \gamma, cc_{\mathcal{P}})$ from $\mathcal{P} \in \gamma.\text{end-users}$ in time at least $\gamma.\text{validity} + 4\Delta + 5$ where $oc_{\mathcal{P}}$ is an opening certificate of $\gamma.\text{Ingrid}$ on γ and γ has not been marked as closed: send a message $(\text{vc-closing}, \gamma.\text{id})$ to $\gamma.\text{Ingrid}$ and wait for one of the following messages:
 - a) Upon receiving $(\text{vc-already-closed}, \gamma, cc_{\mathcal{P}})$ from $\gamma.\text{Ingrid}$, where $cc_{\mathcal{P}}$ is a closing certificate of $\beta.\text{other-party}(\mathcal{P})$ on γ : then do nothing.
 - b) $(\text{vc-close-timeout}, \gamma)$ from \mathcal{P} in time at least Δ after you sent the $(\text{vc-closing}, \gamma.\text{id})$ message to $\gamma.\text{Ingrid}$: then in this case go to subroutine (C) below.
 4. Upon receiving $(\text{lc-close}, W)$ from \mathcal{P} , where $W = (\gamma_{\mathcal{P}}, w_{\mathcal{P}}, \varepsilon, \sigma)$ is a version of β signed by $\mathcal{P}' = \beta.\text{other-party}(\mathcal{P})$: send a message (lc-closing) to \mathcal{P}' and wait for one of the following to happen:
 - a) Upon receiving a rely of \mathcal{P}' with $(\text{lc-close}, W')$ where W' is a version of β signed by $\mathcal{P}' = \beta.\text{other-party}(\mathcal{P})$: let $\text{balance} := \text{Win}(W, W') + \text{transfers}$. For $\hat{\mathcal{P}} \in \beta.\text{end-users}$ send $\text{balance}(\hat{\mathcal{P}})$ coins to $\hat{\mathcal{P}}$'s account on the ledger together with a message (lc-closed) , and close the contract.
 - b) In time τ party \mathcal{P}' replies with a message $(\text{vc-active}, z)$, where z is an opening certificate of \mathcal{P} on some channel γ constructed over β and $\tau \leq \gamma.\text{validity} + 7\Delta + 5$: then do nothing.

- c) Upon receiving (`lc-close-timeout`) from \mathcal{P} in time Δ after you sent the (`lc-closing`) message to \mathcal{P}' : then let $balance := \gamma_{\mathcal{P}}.cash + transfers$. For $\hat{\mathcal{P}} \in \beta.end\text{-users}$ send $balance(\hat{\mathcal{P}})$ coins to $\hat{\mathcal{P}}$'s account on the ledger together with a message (`lc-closed`), and close this contract instance.

(C) Subroutine for closing a virtual channel when cheating by party \mathcal{P} is detected

Let $x := \gamma.cash(\gamma.Alice) + \gamma.cash(\gamma.Bob)$. Remove x coins from \mathcal{P} 's account in *transfer* and add x coins to $\beta.other\text{-party}(\mathcal{P})$'s account in *transfer*. Mark γ as closed. Send a message (`vc-closed`) to both $\beta.end\text{-users}$.

The assumption that for every channel δ each party \mathcal{P} can send at most one message of a given type that concerns δ (see subroutine (B)), is a technical restriction that simplifies the presentation. By message type, we mean the keyword that starts the message. It essentially means that, e.g., no party can ask to close the same channel twice.

4.4.2 The Π_{channel} protocol

Now that we have seen how the channel contract works, we can construct the protocol formally. The overall PERUN Protocol consists of many sub-protocols:

- (A) Ledger channel opening.** This step is performed on-chain with the help of the smart contract functionality \mathcal{C} . If both parties agree they open a ledger channel β and lock the necessary coins for this procedure in the ledger.
- (B) Channel update.** As long as both $\delta.Alice$ and $\delta.Bob$ agree, this step does not require interaction with the contract functionality or Ingrid and can be repeated as many times as necessary. δ can be both a ledger or virtual channel.
- (C) Ledger channel closing.** Any channel party can initiate the closing on-chain via the smart contract functionality \mathcal{C} . It guarantees that the balances of the last agreed upon update will be paid out.
- (D) Virtual channel opening.** This step is performed off-chain with the help of Ingrid. If all three parties agree they open a virtual channel δ over two sub-

4 Virtual Payment Channel Hubs

channels β_A and β_B and lock the necessary coins for this procedure in the subchannels.

(E) Virtual channel closing. In the optimistic case, a virtual channel is closed once its validity expires. This requires interaction of Alice, Bob and Ingrid and makes sure that the virtual channel coins are distributed correctly and added back to the balances of the ledger channels.

Channel Opening Sub-Protocol

This sub-procedure describes the opening of ledger channels β between the parties β .Alice and β .Bob in interaction with the smart contract functionality $\mathcal{C} = \mathcal{C}(\beta.\text{id})$. To open a channel β , party β .Alice sends to \mathcal{C} a contract constructor message for $\mathcal{C}(\beta.\text{id})$ together with $\beta.\text{cash}(\beta.\text{Alice})$ coins. This is a message with a fixed timeout, meaning that β .Alice sends a timeout message if she does not receive a reply from $\mathcal{C}(\beta.\text{id})$ within time Δ after the contract instance $\mathcal{C}(\beta.\text{id})$ appeared on the ledger. In this case β .Alice gets her coins back. The contract sends a message $(\text{1c-opening}, \beta)$ to β .Bob informing him about the fact that β .Alice initiated ledger channel opening. Once the contract gets the confirmation message 1c-open from β .Bob (together with Bob's coins) then the channel is opened.

Protocol Π_{channel} (A) Opening ledger channel β
<ol style="list-style-type: none"> 1. Upon receiving a message $(\text{1c-open}, \beta)$ from the environment, party β.Alice forwards this message to \mathcal{C} and goes to step 3. 2. Upon receiving a message $(\text{1c-open}, \beta)$ from the environment and $(\text{1c-opening}, \beta)$ from $\mathcal{C}(\beta.\text{id})$ (within Δ rounds), party β.Bob replies to $\mathcal{C}(\beta.\text{id})$ with a message $(\text{1c-open}, \beta)$ and goes to step 3. 3. Each party $\mathcal{P} \in \beta.\text{end-users}$ waits for one of the following: <ol style="list-style-type: none"> a) Upon receiving message (1c-not-opened) from $\mathcal{C}(\beta.\text{id})$ output (1c-not-opened) and go idle. b) Upon receiving message (1c-opened) from $\mathcal{C}(\beta.\text{id})$ output (1c-opened) and run sub-procedures (B), (C) and (D) for channel any virtual channel γ.

Once a ledger channel β was opened it can be updated and closed. Additionally, parties can open, update, and close a virtual channel γ .

Channel Update Sub-Protocol

Next we describe the update process for any open ledger or virtual channel (as both procedures work identically). In order to update a channel δ the parties exchange signed channel versions as defined above.

Protocol Π_{channel} .(B) Update of (ledger or virtual) channel δ

Let δ be an open ledger or virtual channel. Each end-party \mathcal{P} of δ stores the latest version $(\hat{\delta}_{\mathcal{P}}, w_{\mathcal{P}}, \alpha_{\mathcal{P}})$ and a signature of the opposite party σ' on this version for δ . Let $P' = \delta.\text{other-party}(P)$ then \mathcal{P} proceeds as follows:

1. (Initiate Update) Only if no update procedure is going on: Upon receiving message $(\text{update}, \delta.\text{id}, \theta, \alpha)$ s.t. for all $Q \in \delta.\text{end-users}$: $\delta.\text{cash}(Q) + \theta(Q) \geq 0$ from the environment \mathcal{Z} party \mathcal{P} waits for the next \mathcal{P} 's update round of δ . When this round comes \mathcal{P} lets $\tilde{\delta}$ be equal to $\hat{\delta}_{\mathcal{P}}$ except that $\tilde{\delta}.\text{cash} := \hat{\delta}_{\mathcal{P}}.\text{cash} + \theta$. Then she sends a tuple $(\text{updating}, \tilde{\delta}, w_{\mathcal{P}} + 1, \alpha, \hat{\sigma})$ (where $\hat{\sigma}$ is \mathcal{P} 's signature on $(\tilde{\delta}, w_{\mathcal{P}} + 1, \alpha)$) to \mathcal{P}' and goes to Step 4
2. (Receive Update) Upon receiving a correctly signed message $(\text{updating}, \tilde{\delta}, w, \alpha, \hat{\sigma}')$ from the counter party \mathcal{P}' s.t. for all $Q \in \delta.\text{end-users}$: $\delta.\text{cash}(Q) + \theta(Q) \geq 0$ and $w = w_{\mathcal{P}} + 1$, compute $\theta := \tilde{\delta}.\text{cash} - \hat{\delta}_{\mathcal{P}}.\text{cash}$, output $(\text{update-requested}, \beta.\text{id}, \theta, \alpha)$ to the environment \mathcal{Z} and go to Step 3.
3. (Confirm Update) If \mathcal{Z} replies with (update-ok) in the next round, compute signature σ on $(\tilde{\delta}, w, \alpha)$, send a message $(\text{update-ok}, (\tilde{\delta}, w, \alpha, \hat{\sigma}))$ to \mathcal{P}' , overwrite the stored version with $(\tilde{\delta}, w, \alpha)$ and the signature with $\hat{\sigma}$, and stop the update procedure. If \mathcal{Z} does not send the update-ok message in the expected round, send a tuple $(\text{updating}, \delta, w + 1, \tilde{\alpha}, \tilde{\sigma})$ to \mathcal{P}' (where $\tilde{\sigma}$ is \mathcal{P} 's signature on $(\delta, w + 1, \tilde{\alpha})$) and $\tilde{\alpha}$ is the update rejection annotation and goes to Step 5
4. (Finalize Update) Upon receiving a message m within 2 rounds (where the message contains a correct signature of \mathcal{P}' on all message parameters), proceed as follows:
 - If $m = (\text{update-ok}, (\tilde{\delta}_{\mathcal{P}}, w_{\mathcal{P}} + 1, \alpha, \hat{\sigma}))$ output (updated) to \mathcal{Z} , overwrite the stored version with $(\tilde{\delta}_{\mathcal{P}}, w_{\mathcal{P}} + 1, \alpha)$ and the signature with $\hat{\sigma}$, and stop the update procedure.

4 Virtual Payment Channel Hubs

- If $m = (\text{updating}, \delta, w_{\mathcal{P}} + 2, \tilde{\alpha}, \tilde{\sigma})$ output **(not-updated)** to \mathcal{Z} , overwrite the stored version with $(\delta, w_{\mathcal{P}} + 2, \tilde{\alpha})$ and the signature with $\tilde{\sigma}$. Then compute signature σ on $(\delta, w_{\mathcal{P}} + 2, \tilde{\alpha})$ and send a message **(update-ok, $(\delta, w_{\mathcal{P}} + 2, \tilde{\alpha}, \sigma)$)** to \mathcal{P}' . Then output **(not-updated)** to \mathcal{Z} and stop the update procedure.
5. (Finalize Failed Update) Upon receiving msg **(update-ok, $(\delta_{\mathcal{P}}, w_{\mathcal{P}} + 2, \alpha, \tilde{\sigma}')$)** (where the tuple is correctly signed by \mathcal{P}') output **(not-updated)** to \mathcal{Z} , overwrite the stored version with $(\delta, w_{\mathcal{P}} + 2)$ and the signature with $\tilde{\sigma}'$, and stop the update procedure.

Channel updating is done with messages **updating** and **update-ok**. Note that channel updating can take up to 5 rounds: In the first round, \mathcal{A} may propose an update. If she does, \mathcal{B} informs the environment about this in the second round and received confirmation in the third, which will make him send a valid signature to \mathcal{A} . Thus, as a result \mathcal{A} outputs either **updated** to \mathcal{Z} if this message was received or **not-updated** otherwise. If \mathcal{B} does not receive an ok from the Environment in the third round, he will propose a new update reverting the channel balance to the initial γ but increasing the version number w . This ensures that \mathcal{B} does not have a signature of \mathcal{A} on a valid update to $\tilde{\gamma}$ and could thus enforce this version on-chain, while \mathcal{A} does not have this power. Thus, in the rejection case, the protocol requires \mathcal{A} to sign the not-updated γ with a higher version $w_{\mathcal{P}} + 2$ in the fifth round.

Channel Closing Sub-Protocol

Protocol $\Pi_{\text{channel}} \cdot (\text{C})$ Closing the ledger channel with identifier id

Each end-party \mathcal{P} of δ stores the latest version $(\hat{\delta}_{\mathcal{P}}, w_{\mathcal{P}}, \alpha)$ and a signature of the opposite party σ' on this version for every open (ledger or virtual) channel δ . Let $P' = \delta.\text{other-party}(P)$ then \mathcal{P} proceeds as follows:

1. Upon receiving a message **(lc-close, id)** (where id is an identifier of some ledger channel β) from the environment party \mathcal{P} lets $(\beta, 0)$ be the initial version of the channel with identifier id and lets V be the last signed version of β which \mathcal{P} received from $\mathcal{P}' = \beta.\text{other-party}(\mathcal{P})$ (if \mathcal{P} has never received such a version then she lets $V = (\beta, 0, \varepsilon, \perp)$). She sends to $\mathcal{C}(id)$ a message **(lc-close, V)**.
2. Upon receiving (in some round τ) a message **(lc-closing)** from $\mathcal{C}(\beta.id)$ party \mathcal{P} does the following:

4 Virtual Payment Channel Hubs

- a) If earlier she received an opening certificate $oc_{\mathcal{P}'}$ of $P' := \beta.\text{other-party}(P)$ on some virtual channel γ that is constructed over virtual channel β and $\gamma.\text{validity} + 7\Delta + 5 > \tau$ — then she sends to $\mathcal{C}(\beta.\text{id})$ a message $(\text{vc-active}, \gamma, oc_{\mathcal{P}'})$ and she continues waiting.
 - b) Otherwise she sends to $\mathcal{C}(\beta.\text{id})$ a message $(\text{lc-close}, W')$, where W' is the last signed version of β that she received from \mathcal{P} (if she has never received such a version then she lets $V' = (\beta, 0, \varepsilon, \perp)$). Upon receiving a message (lc-closed) from $\mathcal{C}(\beta.\text{id})$ she outputs (lc-closed) and goes idle.
3. Upon receiving a message (lc-closed) from $\mathcal{C}(\beta.\text{id})$ party \mathcal{P} outputs (lc-closed) and goes idle.

Closing a ledger channel is performed, with \mathcal{P} and \mathcal{P}' playing roles of Alice and Bob, and messages $(\text{lc-close}, W)$ and $(\text{lc-close}, W')$ corresponding to $W_{\mathcal{A}}$ and $W_{\mathcal{B}}$. Message lc-closing is used by $\mathcal{C}(\beta.\text{id})$ to communicate to party \mathcal{P}' that \mathcal{P} requested channel closing. Message vc-active is used to communicate to the contract (in Step 2b) that there is a virtual channel still open over the ledger channel β . This is handled by the contract in Step 4b). Note that in the optimistic case this procedure takes time 2Δ (one Δ for proposing the closing, and one for confirming). In the pessimistic case it takes 3Δ since \mathcal{P} sends the (τ) message the latest in time 2Δ , and it takes up to one additional Δ for the contract to process it.

Virtual channel opening.

We now describe the protocol for the virtual channels (cf. Figure 4.3 on p. 52). Recall that a virtual channel γ is built over ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$. Let $C_{\mathcal{A}}$ and $C_{\mathcal{B}}$ be the contract instances corresponding to the ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ (respectively).

We start with the opening procedure. Suppose $\beta_{\mathcal{A}}.\text{Alice}$ and $\beta_{\mathcal{B}}.\text{Bob}$ get instructed to open a virtual channel γ with the initial balance $[\mathcal{A} \mapsto x_{\mathcal{A}}; \mathcal{B} \mapsto x_{\mathcal{B}}]$, and validity v . Assume that the channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ have balances as on Figure 4.2. Recall that opening γ results in changed balances of $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ as illustrated on Figure 4.3). Let us now discuss how this channel opening is realized at the protocol level.

Informally, opening γ is done by letting the parties exchanging *opening certifi-*

4 Virtual Payment Channel Hubs

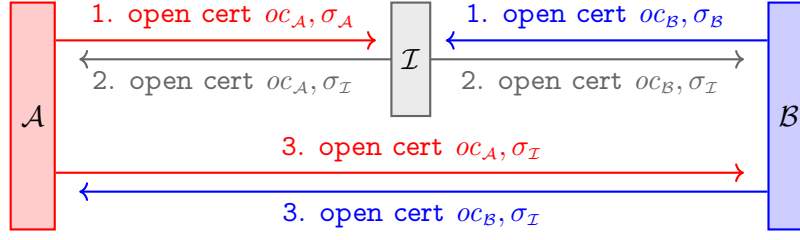


Figure 4.5: Message flow between Alice, Bob and Ingrid during opening of γ .

certificates for γ . Recall, that such an opening certificate of $\mathcal{P} \in \{\mathcal{A}, \mathcal{I}, \mathcal{B}\}$ for γ has the form $(oc_{\mathcal{P}}, \sigma_{\mathcal{P}})$. The role of this certificate is to guarantee that a party \mathcal{P} cannot deny that she agreed to open γ towards the contract instances. For example, if Ingrid denies that she ever agreed to open γ then Alice can use these certificates to prove her wrong during the channel closing (see Section 4.4.2).

Let us first describe the process of opening a virtual channel in case all parties are honest. First, \mathcal{A} and \mathcal{B} send their opening certificates for γ to \mathcal{I} . If \mathcal{I} receives *both* of these certificates, then she replies to \mathcal{A} and \mathcal{B} with her opening certificate for γ and considers the channel open. Parties \mathcal{A} and \mathcal{B} upon receiving Ingrid's certificates forward them to each other. They consider the channel to be open (either upon receiving Ingrid's opening certificate directly from her or receiving it from the channel partner). Pictorially, the message flow looks in this case is depicted in Figure 4.5. Note that the ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ are *not* updated in this procedure. Therefore, technically, virtual channel opening does not result in immediate direct removal of coins from parties' accounts in the ledger channels. Instead the parties locally keep track of the coins (and their distribution) in the ledger channels and remove coins for opening of the virtual channel locally. If necessary, they can always enforce this removal in the on-chain contract. For the security, it is crucial, that the parties locally keep track of all open channels, their certificates and their (latest) versions.

Now consider what happens if some parties are misbehaving. In this case the execution of the protocol can result in *not* opening channel γ . Let us first discuss how our protocol ensures that honest parties will agree on open channels. The result of the protocol execution is that the parties receive opening certificates from the other channel participants for γ . Since such a certificate can later be used to claim coins from a party $\mathcal{P} \in \{\mathcal{A}, \mathcal{I}, \mathcal{B}\}$, the main security risk for \mathcal{P} is that it sends out a (signed) certificate without receiving a (signed) certificate back. In such a case, the other parties might later claim that γ was opened, while \mathcal{P} cannot prove the same. It is easy to see that this problem cannot lead to financial

4 Virtual Payment Channel Hubs

damage for \mathcal{A} and \mathcal{B} . This is because these parties will not consider the channel open if they do not receive an opening certificate from \mathcal{I} , and in this case they will never perform any update to γ . Consider the case where a malicious \mathcal{I} does not send an opening certificate for γ to \mathcal{A} or \mathcal{B} and then requests to close γ when γ 's validity time comes. Note, that her behavior will not change the coin distribution for both \mathcal{A} and \mathcal{B} , just as if the channel was never opened (as the default state of γ is that both parties get the same amount of coins as they deposited). If Ingrid is honest on the other hand, then clearly there is a consensus among all honest parties on whether the channel γ was open or not (since either Ingrid sends her opening certificate to both Alice and Bob, or to none of them). If Ingrid is dishonest, then the only situation when there is disagreement between the *honest* Alice and Bob is if the malicious \mathcal{I} sends her opening certificate to one of them, and not to the other one. To avoid this problem we let the parties forward to each other the opening certificate from \mathcal{I} . This guarantees that if at least one of them considers the channel open, then the other one considers it open as well.

It remains to show that \mathcal{I} stays financially neutral (cf. Section 4.1.1). Here, the problem could potentially be larger, as \mathcal{I} could lose coins if she sends her certificate to Alice (say) without getting the certificate from Bob (as during the channel closing she would be forced to pay coins to Alice *without* being guaranteed that she gets the same amount of coins from Bob). This problem is precisely the reason why in our protocol \mathcal{I} signs the opening certificates *only* if she received the opening certificates for γ from *both* \mathcal{A} and \mathcal{B} . In other words: she only agrees to cover Bob's commitments in front of Alice if she is guaranteed that Bob can be held responsible for these commitments (and symmetrically for Alice's commitments).

Finally, let us comment on the behavior of the parties when the opening procedure successfully ends. One thing that would obviously be dangerous is if one of the parties starts the ledger channel closing procedure for $\beta_{\mathcal{A}}$ or $\beta_{\mathcal{B}}$ when γ is still open (i.e., before its validity time comes). Therefore, after every successful opening of a virtual channel γ , each party $\mathcal{P} \in \{\mathcal{A}, \mathcal{I}, \mathcal{B}\}$ monitors the situation in the ledger channels, and reacts to it. Suppose, for example, that a malicious Ingrid contacts $C_{\mathcal{A}}$ with a request to close channel $\beta_{\mathcal{A}}$ while γ is still open. As described above, $C_{\mathcal{A}}$ informs Alice about this fact. Alice then has a chance to stop the closing of $\beta_{\mathcal{A}}$ by sending to $C_{\mathcal{A}}$ the opening certificate of Ingrid for γ .

Protocol Π_{channel} .(D) Opening virtual channel γ

1. Upon receiving a message $(\text{vc-open}, \gamma)$ from the environment each party $\mathcal{P} \in \gamma.\text{end-users}$ signs and sends her opening certificate $oc_{\mathcal{P}}$ on γ to $\gamma.\text{Ingrid}$, waits one round and goes to Step 3
2. Upon receiving a message $(\text{vc-open}, \gamma)$ from the environment party $\gamma.\text{Ingrid}$ waits one round.
 - a) If she receives (correctly signed) opening certificates $oc_{\mathcal{P}}, oc_{\mathcal{P}'}$ of both $(P, P') = \gamma.\text{end-users}$, she replies to \mathcal{P} and \mathcal{P}' by signing and sending her opening certificate $oc_{\mathcal{I}}$. Then she outputs (vc-opened) , waits until round $\gamma.\text{validity}$ and then goes to the (B) *Virtual channel closing* procedure.
 - b) Otherwise: she outputs (vc-not-opened) and stops.
3. If a party $\mathcal{P} \in \gamma.\text{end-users}$ receives a (correctly signed) opening certificate $oc_{\mathcal{I}}$ on γ from $\gamma.\text{Ingrid}$ then she forwards this certificate to $P' = \gamma.\text{other-party}(P)$, outputs (vc-opened) . Now they can proceed with updates on the virtual channel subprocedure (B) and close the virtual channel at time $\gamma.\text{validity}$ through subprocedure (E).

Virtual channel closing.

The channel closing procedure is started automatically when the validity of γ expires. The main idea of this procedure is that it is \mathcal{I} who is responsible for closing γ and taking care that the channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ are updated in the correct way (i.e., according to the the latest balance of γ). Therefore, in some sense, \mathcal{I} plays a role similar to the role of C for the ledger channel closing. Of course, the situation is more complicated now, since (unlike C), \mathcal{I} cannot be assumed to be trusted.

Our closing protocol is constructed in such a way that it is guaranteed that an honest Ingrid will always manage to close a virtual channel within some fixed time T_{max} (or at least convince the contract that she correctly started the closing procedure). If Ingrid does not close γ on time, this is a uniquely attributable fault, i.e., a contract instance (e.g., $C_{\mathcal{A}}$) can always determine if it was indeed Ingrid who did not close the channel γ , or if Alice is falsely accusing Ingrid. This comes from (i) the fact that Ingrid agreed on opening a channel γ (with validity $\gamma.\text{validity}$) can be proven using the opening certificate oc_{Ingrid} , and (ii) proving that a channel has been closed is possible thanks to the *closing certificates* that we define below.

Therefore, what remains is to describe the protocol in which Ingrid can close γ

4 Virtual Payment Channel Hubs

in bounded time. If this does not happen, then Alice and Bob complain to the contract instances C_A and C_B respectively, and these instances will punish Ingrid by transferring all of Ingrid's coins to the complaining party. If everybody is honest then the procedure works in a straightforward way. Let us start by explaining it, and ignoring for a moment some details that are needed for preventing cheating by dishonest parties. First, Alice sends Ingrid the latest update message V_B that she received from Bob (if no update has been performed then V_B is the initial channel balance of γ with version number 0, and no signature). In parallel, Bob mirrors this behavior with the latest update message V_A that he received from Alice. Then Ingrid decides which of the versions is the latest balance of γ by checking which version has a higher number (this is done according to the same rules as the ones used by C in the ledger channel closing procedure). She then proposes to update the ledger channels accordingly. That is: if the latest balance of γ is $[\mathcal{A} \mapsto x'_A; \mathcal{B} \mapsto x'_B]$ then the balance of the ledger channel β_A is changed to by adding $-x_A + x'_A$ coins to Alice's account and $-x_B + x'_B$ coins to Ingrid's account in β_A (note that these two values sum up to 0), and, symmetrically: adding $-x_A + x'_A$ coins to Ingrid's account and $-x_B + x'_B$ coins to Alice's account in β_B (see also Figure 4.4 on p. 53). Recall that $-x_A$ and $-x_B$ coins were subtracted during the channel opening procedure. Alice and Bob confirm the update, and channel γ is closed.

One problem with the above procedure is that the parties end up with no proof that the virtual channel has been closed. In particular, this means that a dishonest party could later try to close γ again, or Alice and Bob could accuse Ingrid of not closing γ on time. To fix this, we make the following change in the ledger channel update procedure. Instead of exchanging signatures on message m_{β_A} of a form as in Figure (4.4), Alice and Ingrid exchange *closing certificates* $cc_{\beta_A}^\gamma$ on γ defined as

$$m_{\beta_A}^* = (\text{update } \beta_A \text{ to } [\mathcal{A} \mapsto x'_A, \text{Ingrid} \mapsto x'_B] \\ \text{because of closing } \gamma, \text{ version number } w_A), \quad (4.1)$$

where w_A is the current version number used for updating channel β_A . Symmetrically, Ingrid and Bob exchange signatures on the analogously defined $m_{\beta_B}^*$. Hence, a successful closing procedure of γ results in each party holding a signed string that can serve as a proof that γ was correctly closed. We call such signed strings *closing certificates* (cc).

Another problem is that Ingrid has no proof that one of the parties, Alice, say, indeed sent to her the message V_B . In particular, since this message does not contain Alice's signature, it can be easily fabricated by malicious Ingrid collaborating

4 Virtual Payment Channel Hubs

with malicious Bob. Hence, it cannot be later serve as proof from Ingrid during the interaction with the contract instance C_A . We solve this problem by requiring that this message has to come with Alice's signature (and, symmetrically V_A sent by Alice, has to come with Bob's signature). Let $msg_{\mathcal{P}}^i$ (for $i = 1, 2, 3$ and $\mathcal{P} \in \{A, B\}$) denote the consecutive messages that should be exchanged between the parties (if all of them are honest), i.e., $msg_{\mathcal{P}}^1 := (V_{\mathcal{P}'}, \sigma_{\mathcal{P}})$ signed by \mathcal{P} , and $msg_{\mathcal{P}}^2 := (m_{\beta_{\mathcal{P}}}^*, \sigma_{\mathcal{I}})$ signed by \mathcal{I} , and $msg_{\mathcal{P}}^3 := (m_{\beta_{\mathcal{P}}}^*, \sigma_{\mathcal{P}})$ signed by \mathcal{P} . To summarize, the message flow in the closing procedure (in case everybody behaves honestly) looks as depicted on Fig. 4.6.

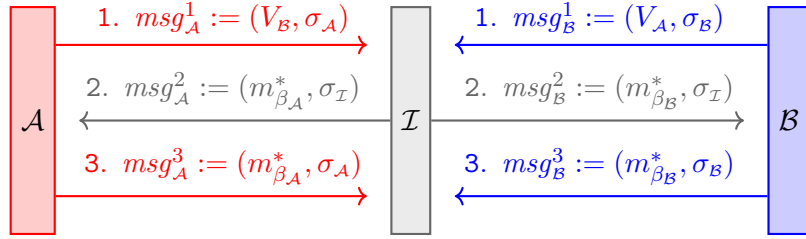


Figure 4.6: Message flow between Alice, Bob and Ingrid during closing of γ .

Consider now what happens when the parties are malicious. Look, e.g., at the interaction between Alice and Ingrid (the interaction between Ingrid and Bob is handled analogously). First, suppose Alice is dishonest and does not send a message msg_A^1 or msg_A^3 to Ingrid. In this case Ingrid has to resolve this issue by contacting C_A . Here the absence of a message is a *non-unique*ly attributable fault i.e., C_A has no way to determine if Alice in fact did not send this message or Ingrid only claims this. Therefore, Ingrid cannot expect C_A to punish Alice immediately. The procedure works as follows. In both cases (msg_A^1 not sent and msg_A^3 not sent) Ingrid initiates her conversation with C_A by providing evidence that Alice should send a message to her. This evidence is different in each case.

msg_A^1 not sent: In this case it is enough that Ingrid sends Alice's opening certificate oc_A for γ to the contract functionality C_A . C_A then checks γ 's validity and rejects the complaint if the channel is still valid, i.e., if the validity did not expire yet. Otherwise C_A informs Alice about Ingrid's complaint. If γ has already been closed then Alice proves it to C_A by replying with a closing certificate on γ signed by Ingrid (in which case C_A punishes Ingrid). Otherwise, Alice sends msg_A^1 to C_A who forwards it to Ingrid and we say that Ingrid received msg_A^1 via the contract. If Alice does not react within time Δ , then it is a uniquely attributable fault, and C_A punishes Alice.

4 Virtual Payment Channel Hubs

$msg_{\mathcal{A}}^3$ *not sent*: In this case Ingrid publishes Alice's opening certificate oc_{Alice} for γ , plus messages msg_1^A and msg_1^B that Ingrid received earlier (either directly from the Alice and Bob, or via the contract C_B). Note that these messages consist of versions of γ signed by Alice and Bob, and hence $C_{\mathcal{A}}$ can determine the final balance $[\mathcal{A} \mapsto x'_{\mathcal{A}}; \mathcal{B} \mapsto x'_{\mathcal{B}}]$ of γ . Since the opening certificate contains the initial balance $[\mathcal{A} \mapsto x_{\mathcal{A}}; \mathcal{B} \mapsto x_{\mathcal{B}}]$ of γ , C can compute the value $q := -x_{\mathcal{B}} + x'_{\mathcal{B}}$. This value corresponds to the coins that should be transferred from Alice to Ingrid (note that x can be negative, in which case $-x$ coins are transferred from Ingrid to Alice).

Ingrid then starts the following emergency closing procedure of channel $\beta_{\mathcal{A}}$. *Closing of $\beta_{\mathcal{A}}$ with simultaneous transfer of q coins from Alice to Bob*: The channel is closed exactly as described in Sect. 4.1.1 except that the amounts of coins that the parties get are corrected to take into account the transfer q . To be more concrete, suppose Ingrid played the role of Bob in channel $\beta_{\mathcal{A}}$ (and Alice played the role of Alice). Let the message $m_{\mathcal{P}^{\mathcal{A}}}^{\beta_{\mathcal{A}}}$ with the higher version number be as on Figure 4.4. Then the amount of coins that Alice gets is $x_{\mathcal{A}} - q$ and Bob (who is Ingrid in our case) gets $x_{\mathcal{B}} + q$.

A less complicated case occurs when Ingrid does not send $msg_{\mathcal{A}}^2$ to Alice, or send a wrong message $msg_{\mathcal{A}}^2$ (e.g., a message that proposes to Alice fewer coins than what she is supposed to receive from closing γ). In this situation, Alice simply does nothing until time T_{\max} comes, or until she gets some message from $C_{\mathcal{A}}$ triggered by Ingrid's action (see above). This is acceptable, as we place the burden to close γ before time T_{\max} on Ingrid.

Protocol $\Pi_{\text{channel}} \cdot (\text{E})$ Virtual channel closing

For $\mathcal{P} \in \gamma.\text{end-users}$ let $\beta_{\mathcal{P}}$ denote the channel with identifier $\gamma.\text{subchan}(\mathcal{P})$, and let $(\gamma_0, 0)$ be the initial version of channel γ . For $\mathcal{P} \in \gamma.\text{all-users}$ let $oc_{\mathcal{P}}$ denote the opening certificate of \mathcal{P} on γ . If $P \in \gamma.\text{end-users}$ then \mathcal{P}' denotes $\gamma.\text{other-party}(P)$.

1. In round $\gamma.\text{validity}$ each $\mathcal{P} \in \gamma.\text{end-users}$ lets $V_{\mathcal{P}'} := (\gamma_{\mathcal{P}'}, w_{\mathcal{P}'}, \alpha_{\mathcal{P}'}, \sigma_{\mathcal{P}'})$ be the latest signed version of γ that \mathcal{P} received from \mathcal{P}' . If \mathcal{P} never received a signed version of γ from \mathcal{P}' (which means that no updates of γ have been performed) then \mathcal{P} lets $V_{\mathcal{P}'} := (\gamma_0, 0, \varepsilon, \perp)$. Then \mathcal{P} sends to $\gamma.\text{Ingrid}$ a tuple $(\text{vc-close}, V_{\mathcal{P}'}, \text{Sign}_{\mathcal{P}}(V_{\mathcal{P}'}))$ and goes to Step 4.
2. In round $\gamma.\text{validity} + 1$ party $\gamma.\text{Ingrid}$ does the following for each $\mathcal{P} \in \gamma.\text{end-users}$:

4 Virtual Payment Channel Hubs

- a) If she receives a correctly formatted $(\mathbf{vc-close}, V_{\mathcal{P}'}, S_{\mathcal{P}})$ message from \mathcal{P} then she goes to Step 3.
 - b) Otherwise she sends a message $(\mathbf{vc-close-init}, \gamma, oc_{\mathcal{P}})$ to $\mathcal{C}(\beta_{\mathcal{P}}.id)$. If she then receives a message $(\mathbf{vc-close}, V_{\mathcal{P}'}, S_{\mathcal{P}})$ from $\mathcal{C}(\beta_{\mathcal{P}}.id)$ then she goes to Step 3. Otherwise she receives a message $(\mathbf{vc-closed})$ — in this case she sets $V_{\mathcal{P}'} := (\gamma_0, 0, \varepsilon, \perp)$ and $S_{\mathcal{P}} := \perp$ and goes to Step 3.
3. Party $\gamma.Ingrid$ waits to learn $(V_{\mathcal{P}'}, S_{\mathcal{P}})$ for *both* $\mathcal{P} \in \gamma.end-users$ (either by getting $(V_{\mathcal{P}'}, S_{\mathcal{P}})$ directly from a party, or via the contract in Step 2b). She lets $\theta := \mathbf{Win}(V_{\gamma.Alice}, V_{\gamma.Bob})$. Then for each $\mathcal{P} \in \gamma.end-users$ she proposes an update of $\beta_{\mathcal{P}}$ that adds $x := \theta(P) - \gamma_0.cash(P)$ coins to \mathcal{P} 's account and $-x$ coins to $\gamma.Ingrid$'s account and is annotated with a string **channel $\gamma.id$ closed**, and goes to Step 5.
 4. Party $\mathcal{P} \in \gamma.end-users$ waits for one of the following events to happen:
 - a) Party $\gamma.Ingrid$ proposes an update to ledger channel $\beta_{\mathcal{P}}$ that adds $\gamma_{\mathcal{P}'}.cash(P) - \gamma_0.cash(P)$ coins to \mathcal{P} 's account and is annotated with a string **channel $\gamma.id$ closed**: \mathcal{P} confirms this update, outputs $(\mathbf{vc-closed})$ and goes to Step 6.
(In case \mathcal{P} in the past did not receive a confirmation on her last update message $(\mathbf{updating}, (\hat{\gamma}, \hat{w}, \hat{\alpha}, \hat{\sigma}))$ she also accepts updates that add $\hat{\gamma}.cash(P) - \gamma_0.cash(P)$ coins to her account.)
 - b) Party \mathcal{P} receives a message $(\mathbf{vc-closing}, \gamma.id)$ from $\mathcal{C}(\beta_{\mathcal{P}}.id)$, party \mathcal{P} replies $(\mathbf{vc-closing}, V_{\mathcal{P}'}, \mathbf{Sign}_{\mathcal{P}}(V_{\mathcal{P}}))$ and continues waiting.
 - c) Within round $\gamma.value + 4\Delta + 5$ none of the above happens: \mathcal{P} sends message $(\mathbf{vc-close-timeout}, \gamma, oc_{\gamma.Ingrid})$ to $\mathcal{C}(\beta_{\mathcal{P}}.id)$, outputs $(\mathbf{vc-closed})$ and continues waiting.
 - d) Party \mathcal{P} receives a message $(\mathbf{vc-closed}, \gamma)$ from $\mathcal{C}(id)$: party \mathcal{P} outputs $(\mathbf{vc-closed})$ and goes to Step 6.
 5. For each of the update procedures proposed by her in Step (3) $\gamma.Ingrid$ does the following:
 - a) If the update procedure is successful then she outputs $(\mathbf{vc-closed})$ and goes to Step 6.
 - b) Otherwise she sends message $(\mathbf{vc-close-final}, \gamma, oc_{\mathcal{P}}, (V_{\gamma.Bob}, S_{\gamma.Alice}), (V_{\gamma.Alice}, S_{\gamma.Bob}))$ to $\mathcal{C}(\beta_{\mathcal{P}}.id)$. Once she receives a message $(\mathbf{vc-closed}, \gamma)$ from $\mathcal{C}(\beta_{\mathcal{P}}.id)$ she outputs $(\mathbf{vc-closed})$ and stops this procedure.

6. A party $\mathcal{P} \in \gamma.\text{all-users}$ goes in to an idle state. If at any point later \mathcal{P} receives a message from \mathcal{C} that concerns channel γ then \mathcal{P} answers with $(\text{vc-already-closed}, cc)$, where cc is the closing certificate on γ (see Sect. 4.4.1).

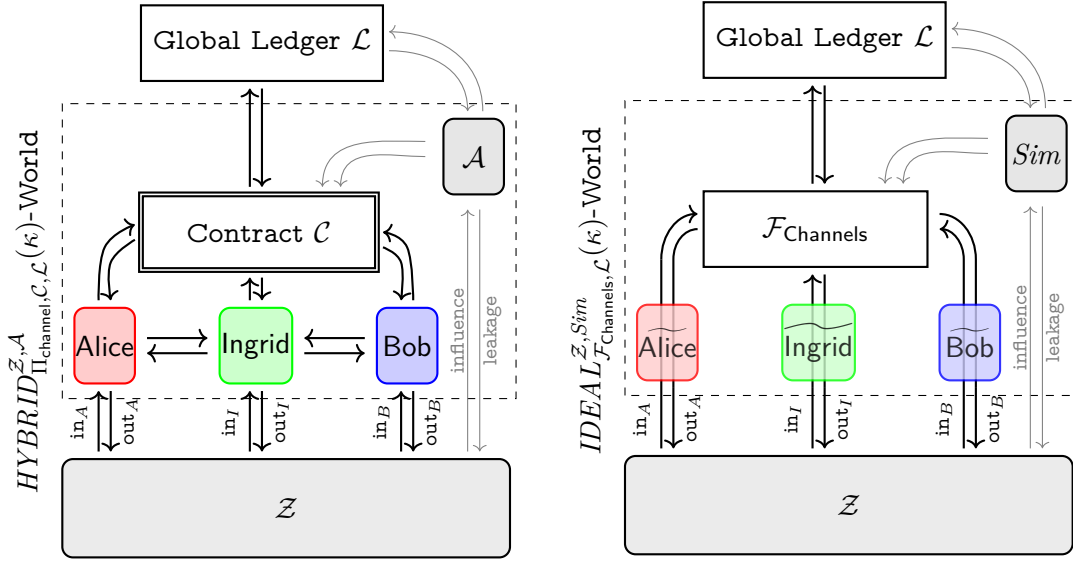
The `vc-close-init` messages sent by the end-users of γ correspond to messages $msg_{\mathcal{P}}^1$ on Fig. 4.6. The updates proposed by $\gamma.\text{Ingrid}$ in Step 3 correspond to messages msg_2^A and msg_2^B . These updates are annotated with strings `channel $\gamma.\text{id}$ closed` since they correspond to messages of a form as on Eq. (4.1) (p. 82). Recall that we considered two cases of malicious behavior of the parties. The actions of $\gamma.\text{Ingrid}$ in the first case ($msg_{\mathcal{P}}^1$ not sent) are described in Step (2b), where $\gamma.\text{Ingrid}$ sends a `vc-close-init` message to the contract. The contract receives this message in Step (1a) and ensures that the virtual channel is closed correctly in time. The second case ($msg_{\mathcal{P}}^3$ not sent) is handled in Step 5. Recall that in our informal description $\gamma.\text{Ingrid}$ had to send \mathcal{P} 's opening certificate on γ and messages $msg_{\mathcal{A}}^1$ and $msg_{\mathcal{B}}^1$ to the contract. In the formal description these values correspond to $oc_{\mathcal{P}}$ and pairs $(V_{\gamma.\text{Bob}}, S_{\gamma.\text{Alice}})$ and $(V_{\gamma.\text{Alice}}, S_{\gamma.\text{Bob}})$ respectively, which are sent to $\mathcal{C}(\beta_{\mathcal{P}}.\text{id})$ in the `vc-close-final` message. Note also that in case \mathcal{P} did not receive a confirmation on her last update message (that contained a channel tuple $\hat{\gamma}$) then she accepts that $\gamma.\text{Ingrid}$ transfers to her the amount of coins that she should get from $\hat{\gamma}$ (and not from $\gamma_{\mathcal{P}'}$). This is needed since $\gamma.\text{Ingrid}$ has no way to find out what happened between \mathcal{P} and \mathcal{P}' when they were updating γ (more concretely: she does not know if indeed \mathcal{P}' did not confirm the update).

4.5 Perun Security Proof

We are now ready to prove Theorem 1 defined in Section 4.3.3. Recall that we need to show that for all PPT adversaries \mathcal{A} , the protocol execution of Π_{channel} in the \mathcal{C} -hybrid world is indistinguishable from the $\mathcal{F}_{\text{channels}}$ world from the view of an environment (which is restricted as described in Section 4.3.1).

We have already informally argued about the security of our scheme while presenting it in the previous sections. Here we focus the formal UC-style proof of security. Figure 4.7 depicts the setup for this proof technique where we distinguish the real and ideal world execution of PERUN (cf. Section 2.4). In both worlds, the environment \mathcal{Z} sends inputs to honest parties and receives outputs from them. In the real world, honest parties behave as described in Section 4.4

4 Virtual Payment Channel Hubs



(a) Hybrid world execution of the PERUN protocol Π_{channel} with Alice, Ingrid, Bob, and \mathcal{A}

(b) Ideal world execution of $\mathcal{F}_{\text{Channels}}$ with dummy parties $\widetilde{\text{Alice}}$, $\widetilde{\text{Ingrid}}$, $\widetilde{\text{Bob}}$ and Sim

Figure 4.7: Setup of a simulation of the PERUN security proof in UC-style manner for honest parties.

and interact with the hybrid contract functionality \mathcal{C} that makes changes in the global ledger functionality \mathcal{L} . In the ideal world, on the other hand, honest parties are so-called dummy-parties that forward any information from and to the environment directly to the ideal functionality $\mathcal{F}_{\text{Channels}}$, which changes the state of the global ledger \mathcal{L} . In the real world execution, the environment also sends instructions to the adversary (called *influence*) and receives information in return (called *leakage*). Recall that we assume a static adversary \mathcal{A} that can corrupt parties at the beginning of the execution. To make both worlds indistinguishable, all changes in the ledger \mathcal{L} , all outputs of honest parties and all leakage of the adversary (towards the environment \mathcal{Z}) in the ideal world must be identical to the ones in the real world. For this purpose, we construct a simulator Sim that interacts with the environment \mathcal{Z} and the ideal functionality $\mathcal{F}_{\text{Channels}}$ on behalf of the adversary \mathcal{A} and all corrupted parties in the ideal world.

At the beginning of the simulation, the simulator Sim internally starts the adversary \mathcal{A} and corrupts the parties that \mathcal{A} would corrupt in the real-world execution. The simulator also generates the (public key, private key) pairs for all parties –

4 Virtual Payment Channel Hubs

even the honest ones. It passes the set of public keys and the private keys of the corrupt users to the adversary (and outputs them to the environment, simulating the leakage of the adversary). Then, *Sim* simulates the behavior of \mathcal{A} and watches the instructions of \mathcal{Z} to the corrupt parties. Depending on these instructions, *Sim* generates messages to simulate the real world protocol execution and sends inputs to the $\mathcal{F}_{\text{Channels}}$ functionality. In particular, he has two tasks to make it impossible to distinguish between the simulated and the real execution. First, the simulator needs to emulate the corrupt parties' outputs, i.e., all messages sent by the \mathcal{C} functionality and other (honest) parties. Moreover, he has to ensure that changes on the ledger \mathcal{L} happen simultaneously in both worlds. Recall also that the adversary \mathcal{A} can control when the ideal functionality $\mathcal{F}_{\text{Channels}}$ processes some message (up to Δ rounds). This accounts for the fact that in the Π_{channel} protocol, the adversary can delay any processing of honest parties' messages to \mathcal{C} by at most Δ rounds. Recall that honest parties always send messages immediately, which ensures that their message reach \mathcal{C} in time before a timeout is triggered. Additionally, corrupt parties may get instructed to send messages to \mathcal{C} at any time they want. Therefore, our simulator has to observe the network and enforce the delays and messages that \mathcal{A} is instructed to introduce also in the ideal world. This ensures that \mathcal{Z} cannot distinguish if \mathcal{C} or $\mathcal{F}_{\text{Channels}}$ made changes in \mathcal{L} . As this is a continuous task of *Sim* in all steps of the protocol, we abstract from it in the sketch of the simulator on to keep the exposition clean.

We proceed by constructing a simulator *Sim* for every step (A) - (E) of the protocol as introduced in Section 4.4.2 and show for each phase why the protocol execution is indistinguishable from that of the ideal functionality. It is easy to see that the only non-trivial cases are when some of the parties participating in a given part of the protocol are corrupt, and some are honest. If all the parties are honest, the protocol proceeds in the fully optimistic case. In this case, the environment only has changes in the global ledger \mathcal{L} and honest parties' outputs as a base for the distinction. In this case, the ideal functionality $\mathcal{F}_{\text{Channels}}$ proceeds in the optimistic case, which trivially emulates the optimistic case execution of the Π_{channel} protocol automatically.

Another edge case scenario occurs if all parties, **Alice**, **Bob**, and **Ingrid**, are corrupted by the adversary. In this case, *Sim* can internally simulate the contract behavior on the input of corrupt parties, and forward any resulting messages to the environment via the leakage. If the parties make the contract functionality \mathcal{C} lock or redistribute coins on the ledger \mathcal{L} , *Sim* can simulate this behavior by updating the balance of corrupted parties in \mathcal{L} using the `init` message (recall that

he has the power to transfer coins from the corrupt parties freely).

4.5.1 Ledger Channel Opening

During this subprocedure, there are only two cases we need to consider: Either the channel is opened (if \mathcal{Z} sent `1c-open` to Alice and Bob) or the channel is not opened (if \mathcal{Z} did not send `1c-open` to one of the parties or one of them is corrupt). This part starts when \mathcal{Z} sends an `(1c-open, β)` message to both β .end-users in some round τ . Simulating it is straightforward: *Sim* simply simulates the contract functionality, plays the role of the honest party to potentially corrupt ones, and ensures that coins are removed from ledger during the right rounds (by delaying messages as discussed above).

Simulator for (A) ledger channel opening

Sim interacts with \mathcal{Z} and $\mathcal{F}_{\text{Channels}}$ in the ideal world. It controls all inputs and outputs of corrupted parties.

For corrupted $\mathcal{P} \in \beta$.end-users

1. Upon being instructed from \mathcal{Z} to send `(1c-open, β)` on behalf of \mathcal{P} , forward this message to $\mathcal{F}_{\text{Channels}}$.
2. Wait to receive a message from $\mathcal{F}_{\text{Channels}}$.
 - a) Upon receiving `(1c-not-opened)` from $\mathcal{F}_{\text{Channels}}$, forward this message to \mathcal{Z} .
 - b) Upon receiving `(1c-opened)` from $\mathcal{F}_{\text{Channels}}$, forward this message to \mathcal{Z} and run simulation of subprocedure (B) and (C) for channel β .

For corrupted β .Bob (additional to the instructions above)

If β .cash(β .Alice) coins are removed in the ledger \mathcal{L} by $\mathcal{F}_{\text{Channels}}$ due to the opening of β , forward message `(1c-opening, β)` to \mathcal{Z} .

In the real world, Bob checks if Alice initiated the channel before he sends his message `(1c-open, β)`, but in the ideal world, this check is performed by $\mathcal{F}_{\text{Channels}}$. Note that when the channel does not open (case 2b), Alice should get her coins back. However, if Alice is corrupted, she could potentially leave the coins in the contract and not request them back. This case is captured by the restriction 4.3.1

4 Virtual Payment Channel Hubs

on the Environment. We could circumvent this restriction by adding some more cases to the descriptions of Π_{channel} , \mathcal{C} , and $\mathcal{F}_{\text{Channels}}$. In the real world, we would let Alice request for a refund (within time Δ after Bob did not react). In the ideal world, we must also be able to capture this case, by letting *Sim* influence the ideal functionality $\mathcal{F}_{\text{Channels}}$ to leave the coins untouched. We would, however, restrict this influence only to be accepted if Alice is corrupted.

4.5.2 Channel Updating

Sim aims to simulate the real-world protocol to \mathcal{Z} in the ideal world while interacting with $\mathcal{F}_{\text{Channels}}$. In the case where neither party is corrupted, the worlds are indistinguishable without *Sim*'s involvement. He only needs to ensure that messages are sent at the right time.

Simulator for (B) channel δ update

Sim interacts with \mathcal{Z} and $\mathcal{F}_{\text{Channels}}$ in the ideal world. It controls all inputs and outputs of corrupted parties.

For corrupted initiators $\mathcal{P} \in \beta.\text{end-users}$

For every open ledger or virtual channel δ the simulator *Sim* stores the latest version $(\hat{\delta}, w_{\mathcal{P}})$.

1. Upon being instructed from \mathcal{Z} to send $(\text{updating}, \hat{\delta}, w_{\mathcal{P}} + 1, \alpha, \hat{\sigma})$ on behalf of \mathcal{P} s, where $\hat{\sigma}$ is a valid signature of \mathcal{P} on the tuple $(\hat{\delta}, w_{\mathcal{P}} + 1, \alpha)$, send message $(\text{update}, \delta.\text{id}, \theta, \alpha)$ to $\mathcal{F}_{\text{Channels}}$, where $\theta := \tilde{\delta}.\text{cash} - \hat{\delta}.\text{cash}$. Then go to step 2.
2. If $\mathcal{F}_{\text{Channels}}$ sends a message (updated) to \mathcal{P} , generate signature σ' of $P' = \delta.\text{other-party}(\mathcal{P})$ on $(\tilde{\delta}_{\mathcal{P}}, w_{\mathcal{P}} + 1, \alpha)$ and output $(\text{update-ok}, (\tilde{\delta}_{\mathcal{P}}, w_{\mathcal{P}} + 1, \alpha, \sigma'))$ to \mathcal{Z} .

For corrupted confirmers $\mathcal{P}' \in \beta.\text{end-users}$

For every open ledger or virtual channel δ the simulator *Sim* stores the latest version $(\hat{\delta}, w_{\mathcal{P}'})$.

1. If $\mathcal{F}_{\text{Channels}}$ sends $(\text{update-requested}, \beta.\text{id}, \theta, \alpha)$ to \mathcal{P}' , compute $\tilde{\delta}$ as $\hat{\delta}_{\mathcal{P}'}$ except $\tilde{\delta}.\text{cash} := \hat{\delta}.\text{cash} + \theta$, generate signature $\hat{\sigma}$ of $P = \delta.\text{other-party}(\mathcal{P}')$ on $(\tilde{\delta}, w_{\mathcal{P}'} + 1, \alpha)$ and output $(\text{updating}, \tilde{\delta}, w_{\mathcal{P}'} + 1, \alpha, \hat{\sigma})$ to \mathcal{Z} .

4 Virtual Payment Channel Hubs

2. Upon being instructed from \mathcal{Z} to send (**update-ok**, $(\tilde{\delta}_{\mathcal{P}}, w_{\mathcal{P}} + 1, \alpha, \sigma')$) on behalf of \mathcal{P}' , within 2 rounds (where the tuple is correctly signed by \mathcal{P}'), send message (**update-ok**) to $\mathcal{F}_{\text{Channels}}$

If the initiator of an update is corrupt, the environment will start sending messages on behalf of this party. In particular, the procedure starts when \mathcal{Z} sends the first protocol message in the name of the update initiator $\mathcal{P} \in \delta.\text{end-users}$. Now *Sim* starts the ideal world execution by sending message (**update**, id, θ, α) on behalf of \mathcal{P} to $\mathcal{F}_{\text{Channels}}$ when \mathcal{Z} sends the first (correctly signed) protocol message.

If in the next round $\mathcal{F}_{\text{Channels}}$ outputs the **updating** request to party $\mathcal{P}' := \delta.\text{other-party}(\mathcal{P})$, which is indistinguishable if \mathcal{P}' is honest. Otherwise, *Sim* simulates message (**updating**, $\hat{\delta}, w_{\mathcal{P}} + 1, \alpha, \hat{\sigma}$) to \mathcal{Z} via the corrupted party. Note, that if \mathcal{P} is honest *Sim* needs to generate his signature for the above message. Recall that *Sim* generates all keys for all parties in the ideal world simulation. Therefore he can compute the signature even of honest parties.

If \mathcal{Z} sends the confirmation message of the protocol (again only if it is correct and signed) *Sim* triggers the update confirmation in the ideal functionality. Then in the next round, *Sim* simulates the confirmation message from \mathcal{P}' if the initiator \mathcal{P} is corrupt. Again, this requires signing messages with \mathcal{P}' private key, but *Sim* can do it since he knows the private keys of all the parties.

4.5.3 Ledger Channel Closing

Ledger channel closing starts when \mathcal{Z} sends to \mathcal{P} (again we call \mathcal{P} the initiator) a message (**lc-close**, id). As in the case of channel opening the simulation is straightforward: the simulator simply simulates the other parties and the contract functionality for corrupt parties, and ensures that the unlocking of coins and the **lc-close** message of $\mathcal{F}_{\text{Channels}}$ is executed in the correct round.

Simulator for (C) ledger channel closing

Sim interacts with \mathcal{Z} and $\mathcal{F}_{\text{Channels}}$ in the ideal world. It controls all inputs and outputs of corrupted parties.

For corrupted closing initiators $\mathcal{P} \in \beta.\text{end-users}$

1. Upon being instructed from \mathcal{Z} to send (**lc-close**, $\beta, v, \varepsilon, \sigma_{\mathcal{P}'}$) on behalf of \mathcal{P} , where $\sigma_{\mathcal{P}'}$ is a valid signature of $\mathcal{P}' = \beta.\text{other-party}(\mathcal{P})$ on (β, v, ε) ,

4 Virtual Payment Channel Hubs

<p style="text-align: center;">send message $(\text{lc-close}, \beta.\text{id})$ to $\mathcal{F}_{\text{Channels}}$.</p> <ol style="list-style-type: none"> 2. Wait to receive a message (lc-closed) from $\mathcal{F}_{\text{Channels}}$ and forward this message to \mathcal{Z}.
For corrupted channel partners $\mathcal{P}' = \beta.\text{other-party}(\mathcal{P})$
<ol style="list-style-type: none"> 1. One round after \mathcal{P} initiated channel closing, simulate \mathcal{C} by sending (lc-closing) to \mathcal{Z} via the corrupted \mathcal{P}'. 2. Upon being instructed from \mathcal{Z} to send $(\text{lc-close}, \beta', w, \varepsilon', \sigma_{\mathcal{P}})$ on behalf of \mathcal{P}', where $\sigma_{\mathcal{P}}$ is a valid signature of $\mathcal{P} = \beta.\text{other-party}(\mathcal{P}')$ on $(\beta', w, \varepsilon')$, send message $(\text{lc-close}, \beta'.\text{id})$ to $\mathcal{F}_{\text{Channels}}$. 3. Wait to receive a message (lc-closed) from $\mathcal{F}_{\text{Channels}}$ and forward this message to \mathcal{Z}.
For both corrupted $\mathcal{A} = \beta.\text{Alice}$ and $\mathcal{B} = \beta.\text{Bob}$ (additional to the instructions above)
<p>Let $\hat{\beta}$ be the version of the ledger channel that \mathcal{Z} sent through the corrupted parties which would be enforced by \mathcal{C}, i.e., $\hat{\beta} = \beta$ is $v > w$ and $\hat{\beta} = \beta'$ otherwise. If \mathcal{F} adds $p_{\mathcal{A}} \neq \hat{\beta}.\text{cash}(\mathcal{A})$ and $p_{\mathcal{B}} \neq \hat{\beta}.\text{cash}(\mathcal{B})$ coins back to the accounts of $\beta.\text{Alice}$ and $\beta.\text{Bob}$ on the ledger, simulate the coin distribution of $\hat{\beta}$ as follows:</p> <ul style="list-style-type: none"> • if $p_{\mathcal{A}} > \hat{\beta}.\text{cash}(\mathcal{A})$, transfer $\hat{\beta}.\text{cash}(\mathcal{B}) - p_{\mathcal{B}}$ coins from \mathcal{A}'s account in ledger to \mathcal{B}'s account. • if $p_{\mathcal{A}} < \hat{\beta}.\text{cash}(\mathcal{A})$, transfer $\hat{\beta}.\text{cash}(\mathcal{A}) - p_{\mathcal{A}}$ coins from \mathcal{B}'s account in ledger to \mathcal{A}'s account.

The only tricky case occurs when \mathcal{Z} instructs both corrupt parties $(P, Q) = \beta.\text{end-users}$ and to enforce an outdated version of a channel. In this case it can happen that a version is enforced that is less beneficial for one user than the last agreed upon version (this can only happen if both parties are corrupted). We simulate this outcome, by adjusting the balances of both users in \mathcal{L} in the ideal world (using the `init` influence for the simulator on behalf of corrupted parties), thus ensuring that \mathcal{Z} cannot distinguish the outcome of the ideal and real world result.

4.5.4 Virtual Channel Opening

The simulation proceeds as in the previous cases, i.e., Sim mimics the behavior of the corrupt parties towards $\mathcal{F}_{\text{Channels}}$ and emulates the protocol behavior for the environment. Again, the honest case where none of the parties is corrupted is straightforward. All messages that the parties get from the environment in the ideal real-world execution are sent to the ideal functionality, and the result is indistinguishable for \mathcal{Z} .

Simulator for (D) virtual channel γ opening

Sim interacts with \mathcal{Z} and $\mathcal{F}_{\text{Channels}}$ in the ideal world. It controls all inputs and outputs of corrupted parties.

For corrupted parties $\mathcal{P} \in \gamma.\text{end-users}$

1. Upon being instructed from \mathcal{Z} to send an opening certificate $oc_{\mathcal{P}}$ on a virtual channel γ with a valid signature on behalf of \mathcal{P} , send message $(\text{vc-open}, \gamma)$ to $\mathcal{F}_{\text{Channels}}$.
2. Upon being notified by $\mathcal{F}_{\text{Channels}}$ that any party $\mathcal{I} = \gamma.\text{Ingrid}$ sent an opening request for channel γ , sign an opening statement over γ in the name of \mathcal{I} and send the resulting opening certificate $oc_{\mathcal{I}}$ to \mathcal{Z} in the name of \mathcal{P} . Repeat this for both end-parties in γ .
3. Upon receiving a message (vc-opened) from $\mathcal{F}_{\text{Channels}}$ for channel γ , simulate a message from the other channel end-party by signing an opening statement over γ in the name of \mathcal{I} and send the resulting opening certificate $oc_{\mathcal{I}}$ to \mathcal{Z} in the name of \mathcal{P} .

For corrupted intermediaries $\mathcal{I} = \gamma.\text{Ingrid}$

1. Upon being notified by $\mathcal{F}_{\text{Channels}}$ that any party $\mathcal{P} \in \gamma.\text{end-users}$ sent an opening request for channel γ , sign an opening statement over γ in the name of this party and send the resulting opening certificate $oc_{\mathcal{P}}$ to \mathcal{Z} in the name of \mathcal{I} . Repeat this for both end-parties in γ .
2. Upon being instructed from \mathcal{Z} to send a (correctly signed) opening certificate $oc_{\mathcal{I}}$ in the name of \mathcal{I} , send message $(\text{vc-open}, \gamma)$ to the ideal functionality $\mathcal{F}_{\text{Channels}}$.

Whenever all honest dummy parties agree to open a virtual channel, Sim sends the $(\text{vc-open}, \gamma)$ message in the name of all corrupted $\mathcal{P} \in \gamma.\text{all-users}$ to the

ideal functionality $\mathcal{F}_{\text{Channels}}$ and lets the ideal functionality immediately output (**vc-opened**) to all the users. This ensures indistinguishability since virtual channels are always opened in the ideal world, when honest parties agree that such channels are opened in the real world.

4.5.5 Virtual Channel Closing

Virtual channel closing is the most complicated case of the protocol. Here we need to distinguish the case when the intermediaries are misbehaving and the case when the end-parties are malicious. Note that again, the case of three honest parties is straightforward.

Simulator for (D) virtual channel γ opening

Sim interacts with \mathcal{Z} and $\mathcal{F}_{\text{Channels}}$ in the ideal world. It controls all inputs and outputs of corrupted parties.

For corrupted intermediaries $\mathcal{I} = \gamma.\text{Ingrid}$

1. For every honest $\mathcal{P} \in \gamma.\text{end-users}$ of γ simulate the initiation message (**vc-close**, $V_{\mathcal{P}'}$, $\text{Sign}_{\mathcal{P}}(V_{\mathcal{P}})$) for where $V_{\mathcal{P}'} = (\gamma, w, \alpha, \sigma_{\mathcal{P}'})$ was either the last valid version that *Sim* received from a corrupted $\mathcal{P}' = \gamma.\text{other-party}(\mathcal{P})$ is a successful update of γ , or if both parts are honest let γ be the latest version that was registered in $\mathcal{F}_{\text{Channels}}$, w is the number of updates of γ , $\alpha = \epsilon$ and $\sigma_{\mathcal{P}'}$ a valid signature of the previous values by \mathcal{P}' .
2. Upon being instructed from \mathcal{Z} to send (**vc-close-init**, $oc_{\mathcal{P}}$) in the name of \mathcal{I} , simulate the response from contract \mathcal{C} by
 - Sending (**vc-closing**, $\gamma.\text{id}$) to any corrupt $\mathcal{P} \in \gamma.\text{end-users}$.
 - If within time Δ \mathcal{Z} sends a correctly signed message (**vc-closing**, $V_{\mathcal{P}'}$, $\text{Sign}_{\mathcal{P}}(V_{\mathcal{P}})$) on behalf of a corrupted party, output (**vc-close**, $V_{\mathcal{P}'}$, $S_{\mathcal{P}}$) to the environment simulating a message from $\mathcal{C}(\beta_{\mathcal{P}}.\text{id})$ for \mathcal{I} .
 - If all other parties are honest, *Sim* needs to simulate this message instead (just as in the first Step of this simulation). Again it sends (**vc-close**, $V_{\mathcal{P}'}$, $S_{\mathcal{P}}$) to the environment simulating a message from $\mathcal{C}(\beta_{\mathcal{P}}.\text{id})$ for \mathcal{I} .
3. Upon being instructed from \mathcal{Z} to send an update request for the subchannels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$, simulate the update procedures as described in simulation (B).

4 Virtual Payment Channel Hubs

4. If corrupted Ingrid did not send any of the messages as defined in step 2 or step 3, simulate the forced timeout by outputting `(vc-closing, γ .id)` to \mathcal{Z} .
 - Upon being instructed from \mathcal{Z} to send `(vc-already-closed, z)` in Δ rounds, where z is a closing certificate of β .other-party(\mathcal{P}) on γ then do nothing.
 - otherwise simulate the message `(vc-closed)` to any corrupt end-party \mathcal{P} .
5. Upon being notified by $\mathcal{F}_{\text{Channels}}$ that any party $\mathcal{P} \in \gamma$.end-users sent an opening request for channel γ , sign an opening statement over γ in the name of this party and send the resulting opening certificate $oc_{\mathcal{P}}$ to \mathcal{Z} in the name of \mathcal{I} . Repeat this for both end-parties in γ .

For corrupted parties $\mathcal{P} \in \gamma$.end-users

1. Upon being instructed from \mathcal{Z} to send `(vc-close, $V_{\mathcal{P}'}, \text{Sign}_{\mathcal{P}}(V_{\mathcal{P}})$)` in round γ .validity on behalf of \mathcal{P} , where $V_{\mathcal{P}'} = (\gamma, w, \alpha, \sigma_{\mathcal{P}'})$ is correctly signed by $\mathcal{P}' = \gamma$.other-party(\mathcal{P}) output this message to the environment if γ .Ingrid is corrupted. Then go to step 3.
2. If \mathcal{Z} did not instruct corrupted party \mathcal{P} to send the expected message in round γ .validity simulate the execution of \mathcal{C} .
 - Send `(vc-closing, γ .id)` to any corrupt $\mathcal{P} \in \gamma$.end-users.
 - If within time Δ \mathcal{Z} instructs a corrupted party \mathcal{P} to send a correctly signed message `(vc-closing, $V_{\mathcal{P}'}, \text{Sign}_{\mathcal{P}}(V_{\mathcal{P}})$)`, wait 1 round and go to step 3.
3. Simulate the update of the underlying ledger channels where the new balance is changed by the outcome of the virtual channel. In this update Ingrid acts as the initiator (see simulation (B)).

Closing of virtual channel starts automatically when time γ .validity comes. As argued in Section 4.4.2, the virtual channel is always closed, as long as at least one party on γ .all-users is honest. Again, Sim simulates the execution of the contract towards corrupt parties and, depending on their behavior, instructs the ideal functionality $\mathcal{F}_{\text{Channels}}$ to send the `(vc-closed)` message to the honest parties (in time at most γ .validity + $7\Delta + 5$).

4.6 Implementation and Performance

In this section, we summarize the performance of the PERUN protocol. This includes the best case and worst case execution times, message, and storage complexities but also the costs for executing the Ethereum smart contracts.

4.6.1 Execution Times

Let us take a look at how much time, pessimistically, γ .Ingrid needs in order to close γ . First, she needs to wait 1 round to receive the **vc-close** messages from both Alice and Bob. If she does not receive any of them, then she needs to let the contract know about it by sending a **vc-close-init** message to the contract. Sending this message takes Δ rounds in the worst case. Then, the end-party has to respond to the contract (which takes another Δ rounds), and if she does not respond then γ .Ingrid sends a (τ) message (another Δ rounds). After these 3Δ rounds, γ .Ingrid initiates a channel update procedure (that takes at most 4 rounds). If this is unsuccessful then she sends a message **vc-close-final** to the contract which takes one more blockchain delay Δ . Hence within time $\gamma + T_{\max}$, where $T_{\max} = 4\Delta + 5$: either γ .Ingrid closed the channel γ , or the contract received a message **vc-close-final**. The end-party either responds to **vc-close-final** with a **vc-already-closed** message, or another (**timeout**) message is needed (which in total takes time 2Δ). If γ .Ingrid did not close γ within time $\gamma + T_{\max}$, then the end-parties have to close it. It is easy to see that it takes time at most 3Δ (Δ rounds for the **vc-close-timeout** message, another blockchain interaction for waiting for γ .Ingrid's response, and yet another one of the (**timeout**) message). Thus, pessimistically, the virtual channel closing takes time $\gamma + T_{\max} + 3\Delta = 7\Delta + 5$. Optimistically, the virtual channel closing procedure takes 5 rounds (1 round for the **vc-close** messages, and 4 rounds for channel update).

4.6.2 Implementation and Gas Costs

For the purpose of measuring these costs, we implemented the PERUN contract and published it at <https://github.com/PerunEthereum/Perun>. In this section, we present the findings and benchmarks. The implementation shows both feasibility and also gives an idea of the fee costs of the channels. While execution fees were not discussed and considered for the formal protocol specification and proof, they influence whether the channels can be built efficiently, and when it makes sense to use them.

4 Virtual Payment Channel Hubs

Protocol phase	Size	Fees		
	[gas]	[gWei]	[ETH]	[EUR]
Deployment	2757111	8271333	0.008271333	1.06981421022
(A) Open LC	62337	187011	0.000187011	0.02418800274
(B) Update LC/VC	0	0	0	0
(C) Close LC (optimistic)	147788	443364	0.000443364	0.05734469976
(C) Close LC (pessimistic)	275049	825147	0.000825147	0.10672451298
(D) Open VC	0	0	0	0
(E) Close VC (optimistic)	0	0	0	0
(E) Close VC (pessimistic)	418318	1254954	0.001254954	0.16231575036

Table 4.1: PERUN execution fees (with exchange rates from Section 3.2.1).

Table 4.1 displays the execution costs of running the PERUN protocol using the `LedgerChannel` contract. The execution costs are dominated by the deployment fees of over 1 euro.

If both Alice and Bob jointly agree to open, update and close the channel (the optimistic case), they need to execute four on-chain transactions and pay less than 0.10 euros (excluding the deployment cost). Specifically, both parties have to send one transaction each for the opening and closing. To measure the costs for disagreement, we always consider the worst possible case with most on-chain transactions and the highest gas costs (pessimistic case). If either Alice or Bob tries to close the ledger channel with an outdated state while a virtual channel is still active (LC close pessimistic), the other party sends a proof of a (newer) version with an open virtual channel to the smart contract. Settling this disagreement in the smart contract raises the costs for both parties to 0.13 euros. If the parties go to the smart contract in order to dispute over the virtual channels, they additionally need to pay 0.11 euros for every open virtual channel. Note that in this case, Ingrid needs to participate in the on-chain dispute on behalf of one of the parties. In the most costly scenario, she needs to request the closing of the virtual channels, then wait for the other party (e.g., Alice) to make a move, and send another transaction to the blockchain to finalize the dispute. This worst-case scenario limits the fees Ingrid can be forced to pay in the most unfortunate outcome. Let us now take a look at the message complexity of our protocol, i.e., the number of messages sent between the parties involved in the protocol. Notice that each such message consists of a subset of two Ethereum addresses, eight integers (three channel ids,

the cash distribution, the validity, and a version number) as well as two signatures over all of these values. The signatures are the dominating factor for both message length and computation complexity.

We note here that the protocol can be adapted slightly to reduce message complexity for both the update and closing procedure. If both parties sign a closing statement, which invalidates all other versions of the channel, then the smart contract only needs this one final witness for an immediate close. This leads to a speedy and cheap closing procedure. Another practical optimization that we want to highlight is that by allowing one party to submit an outdated state to the contract, we can reduce the messages required for an update to a single message. This change requires us to adapt the protocol slightly to allow honest parties to send an outdated version during the closing procedure. This is necessary since the recipient of a message does not counter-sign it. So only one of the two parties has the latest version of the channel.

4.6.3 Channel Network Comparison

For the ledger channel opening and closing procedures, the message complexity is similar to that of existing payment network systems like Lightning [161] and Sprites [148]. The main advantage of PERUN is the fact that the virtual payment channel can be updated instantaneously by the two parties without sending messages to the intermediaries. This means that after a virtual channel is set up, it can be updated without additional delays by sending only two update messages. The new version, new balances, and a signature are sent by the sender, and the receiver responds with a single signature. Sending the same transaction through one relay in HTLC-based systems requires the computation of at least six signatures, and the intermediary has to receive, compute, and send at least two messages. In other systems, this is even higher. The message complexity limits the effective throughput of how many transactions can be sent over such a system per second.

4.7 Discussion and Extension

We introduced an off-chain payment channel system called PERUN. Its main advantage over the existing solutions is that it allows creating *virtual channels*, which are channels of length 2 that do not require interacting with the intermediary for every payment. The security of our protocol is defined in the UC framework and is formally proven. Our work can be generalized in many directions. Longer state

4 Virtual Payment Channel Hubs

channels are described in subsequent work [71]. One can also ask if it is possible to create a scheme in which the intermediaries do not need to block the coins that are used for constructing virtual channels. This can be done by slightly relaxing the security guarantees. Namely, one can replace the full cheating-resilience (that has been assumed in this work), by a weaker notion of cheating-*evidence*. We leave formalizing this as an interesting future direction.

The role of the intermediary. It is interesting to look at the assumptions and trust that the PERUN system puts in the intermediaries. Without their cooperation, new virtual channels cannot be opened. They can also, at any point in time, request the underlying ledger channels to close, which forces the system participants to watch the ledger constantly and react if necessary. If the hub is malicious or unreliable, the users may decide to open new ledger channels directly to each other or move to a different hub, which requires on-chain transactions. However, an honest hub is, to some degree, a safety barrier for honest users. Consider the scenario where an honest Alice has a reliable connection to Ingrid and then decides to open a virtual channel with an untrusted party, Bob. If Bob starts misbehaving in the virtual channel, Alice knows that she only needs to proceed on-chain (and pay transaction fees) if Ingrid acts maliciously. Therefore, Alice is shielded by Ingrid from the potential risk of costly on-chain dispute interactions.

Ingrid is a so-called *non custodial* hub, which means that she has to pay the collateral costs for every virtual channel that is opened over her, i.e., she has to lock the exact balance that the channel end-parties lock in the virtual channel. However, it is important to note, that Ingrid does not control the coins of Alice and Bob, but instead only relays their payments. This collateral overhead guarantees balance security to honest parties. While balance security is also guaranteed for an honest Ingrid, she still has costs for providing the hub service. Therefore, we assume that Ingrid will ask for fees that compensate her for these costs. If multiple hubs compete with each other new users will most likely choose the hub based on their fees, their reliability, and their connectivity. We proved that Intermediaries cannot benefit from misbehavior and cannot harm the users.

4.7.1 Extensions and Impact

Virtual channels have received widespread attention, and we will present here the academic improvements that have been made since the initial publication of [69].

“General state channel networks”

The PERUN virtual channels construction serves as the main inspiration for the paper “General state channel networks” [71], which aims for the same security guarantees but supports more features. The first improvement is that it considers state channels instead of payment channels, i.e., channel end-parties can execute a smart contract off-chain. As long as the two channel end-parties agree, they can add a deterministic two-party contract⁵ in a channel. The coins that are used inside this contract will be locked during the lifetime of the contract and only paid out according to the final distribution that the contract outputs. Now the parties evaluate the contract by sending their contract interaction and an updated contract state off-chain during the channel update process. In case of disputes, the contract needs to be evaluated on-chain, which means the last agreed-upon state is *registered* on the blockchain. But in order to guarantee that parties can continue the game, they can finish the game on-chain through the so-called *force-execution* process.

The second improvement is that it allows longer routing instead of just one-hub hub-based virtual channels. In fact, it generalizes the concept of virtual channels, such that they can be built on top of virtual channels as well. To explain the main extension, let us consider a virtual channel γ^* , which is built on top of two virtual channels, γ_A and γ_B . If now a dispute occurs in the top virtual channel γ^* , then the parties first try to resolve this dispute with the intermediary that connects them. Only if this does not work, the dispute process continues in the ledger channels below. Again, this only requires the misbehaving party to go on-chain while the other link can stay off-chain. However, a downside to this approach is that the cascading dispute process can lead to very long worst-case timings, which influence the timeouts.

“Multi-party Virtual State Channels”

Therefore, a further extension of the virtual channels framework is the paper “Multi-party Virtual State Channels” [68], which proposes a new dispute mechanism. While so far, virtual channels use *indirect dispute*, in which dispute is always escalated to the underlying channels, and the intermediary has to get involved, [68] introduces direct disputes that allow parties in virtual channels to resolve the dispute among each other on-chain.

⁵Such contracts must not depend on any on-chain data, require inputs from any outside participants, or be time-dependent.

4 Virtual Payment Channel Hubs

But the main contribution of this work is that it allows more than two parties to interact through an off-chain channel. An n -party channel connects not only two end-parties but n end-parties. This means each party can lock coins into the channel, and the channel balance is distributed over n accounts. But this means any channel update does require the approval of all channel participants. Recall that the PERUN construction prevents that two parties can propose updates at the same time. If we extend this solution to n parties, the update procedure would require $\mathcal{O}(2n)$ rounds. Instead, we solve both problems by changing the update procedure and allow an update to have inputs from every party, which are evaluated in a deterministic way (using a sorting function f). The update process looks as follows:

1. First, all parties send their inputs to the update.
2. All parties evaluate function f on all received updates.
3. The update (which contains the result of the function evaluation from the previous step) is signed and shared between all channel participants.
4. If all parties receive n valid signatures on the same value, they continue. Otherwise, they start the dispute process.

This update procedure runs in constant time, even when the number of channel participants gets large. Additionally, it guarantees that they achieve a (simplified notion of) consensus, i.e., either they all agree on the same set of inputs and new channel state, or they settle the update through a dispute.

While [148] already proposed multi-party state channels that are built on top of the ledger, in [68], we show that also virtual channels can be between more than two parties. In particular, we can build multi-party virtual channels on top of 2-party channels (ledger or virtual).

The resulting virtual channels framework consisting of the papers “Perun: Virtual Payment Hubs Over Cryptocurrencies” [69], “General state channel networks” [71] and “Multi-party Virtual State Channels” [68] is a very powerful construction. A network that supports all of these protocols allows many parties to set up a channel once, and then potentially connect them off-chain by opening virtual channels (assuming the channel capacities are sufficient). The parties can decide to send payments, execute contracts, and even build multi-party channels with any subset of the network participants. Additionally, all parties can choose between

4 Virtual Payment Channel Hubs

two dispute mechanisms for every new channel. Direct channels allow short dispute timeouts, as any potential dispute is resolved directly on-chain, while indirect dispute escalates the dispute process to the underlying channels.

5 Moving Complex Computation Off-Chain

Summary. This chapter proposes another approach to scaling with off-chain protocols, based on the publication “FairSwap: How To Fairly Exchange Digital Goods” [67]. While payment channels can be employed to reduce the number of on-chain transactions, we now focus less on the transaction quantity but try to reduce their size and complexity instead. The protocol we design in this work is called FAIRSWAP and aims to reduce the gas costs of large and complicated smart contracts (in Ethereum). In particular, we reduce the deployment costs by decreasing the code size, and we limit the execution costs by minimizing contract storage and function sizes. Again, we will utilize an optimistic off-chain protocol, which shifts the task of function evaluation from the contract to the users.

As a motivating example, we consider the setting where a large digital file is sold over the blockchain. Using the smart contract for this problem ensures financial fairness to both the seller and the buyer. In particular, they know that the seller will receive the payment if and only if the buyer received the correct file, where the correctness is checked via a publicly known file hash. The trivial solution for this problem requires storing the whole document on the blockchain, which is extremely expensive for large files.

FAIRSWAP reduces these high gas costs by reducing the complexity and, thus, also the costs for the function evaluation inside the contract. Instead of running the computation itself, the contract only acts as a judge, which holds the coins during the sale and ultimately transfers them to the right user. If the buyer does not receive the correct file, he can prove this to the judge contract by sending a small proof of misbehavior. Our solution guarantees fairness and efficiency, even for large witnesses.

Our FAIRSWAP protocol works for generic function ϕ that verifies a witness x and outputs 1 if the witness is as expected and 0 otherwise. While there have been several proposals for building fair exchange protocols over cryptocurrencies, our solution minimizes the cost of running smart contracts on the blockchain and avoids

expensive cryptographic tools such as zero-knowledge proofs. We provide formal security definitions for smart contract-based fair exchange and prove the security of our construction. Additionally, we evaluate the practicality of FAIRSWAP via a prototype implementation for our motivating example of selling large files over Ethereum.

5.1 Overview

In this section, we start with a high-level overview of our construction before we proceed with the formal construction and security analysis in the following chapters.

Setup. We start by sketching the overall setup of the problem we consider (Figure 5.1). Here, a receiver \mathcal{R} wishes to buy a digital commodity x from a sender \mathcal{S} . The receiver is willing to pay p coins for a digital commodity $x \in \{0, 1\}^{(n \times \lambda)}$ if it is correct, meaning that it satisfies some predicate function $\phi : x^{(n \times \lambda)} \mapsto \{0, 1\}$ (i.e., if $\phi(x) = 1$).

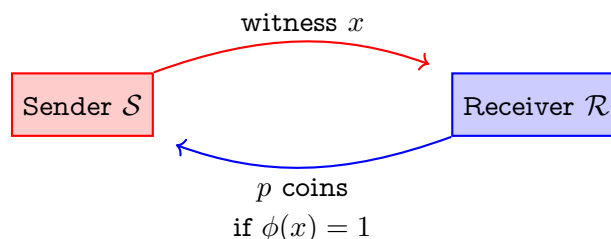


Figure 5.1: Setup for digital fair sale between sender \mathcal{S} and receiver \mathcal{R} .

This fair exchange of digital goods or *contingent payments* in the setting of cryptocurrencies were initially proposed by Maxwell in 2016 [140]. He proposes a solution for the following scenario: \mathcal{R} is willing to pay the first person who can provide a valid solution to a tricky Sudoku puzzle. In this case, x would be the solution, and ϕ specifies the rules of this particular Sudoku game and outputs 1 if the solution is correct. In another example, x could be a large file (e.g., some movie) where the receiver has knowledge¹ of its hash h . In this case, evaluating ϕ requires computing the hash $H(x)$ and comparing the result against h (i.e., $\phi(x) = 1 \iff H(x) = h$). Suppose that the parties wish to execute the exchange

¹The hash could come from a trusted source or could be publicly known.

over the Internet, where \mathcal{R} and \mathcal{S} do not trust each other. A fundamental challenge in this setting is to guarantee that the exchange is fair. In particular, how to ensure that \mathcal{S} receives the payment when he delivers the correct x to \mathcal{R} , and similarly, that \mathcal{R} does not need to pay if x is incorrect (i.e., $\phi(x) \neq h$). Unfortunately, it has been shown that without further assumptions, it is impossible to design protocols that guarantee such strong fairness properties [155].

A simple way to circumvent this impossibility is to introduce a Trusted Third Party (TTP), in practice often referred to as an escrow service [125]. This middleman receives the money from \mathcal{R} and the commodity x from \mathcal{S} . He only executes the exchange if $\phi(x) = 1$ is satisfied. However, such a fully trusted mediator is often not available or very costly. Therefore smart contracts offer an appealing alternative for implementing such an escrow service.

5.1.1 Intuition and Design Ideas

We will now see how to design an efficient fair sale protocol by taking a brief look at three approaches of smart contract-based fair exchange. We start with the straightforward implementation of an escrow contract; next, we analyze an improvement using zero-knowledge proof systems. Finally, we sketch our FAIRSWAP protocol.

Straightforward escrow contract. We start by considering a simple escrow protocol that utilizes a smart contract. In an initial step, the two parties \mathcal{R} and \mathcal{S} set up a contract, where \mathcal{R} blocks p coins. The sender \mathcal{S} now has to send x to the contract within Δ rounds. Otherwise, the money goes back to the receiver \mathcal{R} . If the contract receives the witness, it evaluates $\phi(x)$, and if the output is 1 sends the coins to the sender. If the output is 0, on the other hand, the receiver gets all the coins stored in the contract back. A simplified² version of this protocol is depicted in Figure 5.2.

While the above smart contract example achieves fairness for \mathcal{S} and \mathcal{R} , it has a fundamental drawback if x is large. Since in cryptocurrencies, users pay fees to the miners for every step of a smart contract execution, storing and computing complex instructions results in high fee costs. For instance, in Ethereum, the amount of *gas* paid for executing the smart contract strongly depends on two factors: (a) the complexity of the program ϕ and (b) the size of x . Concretely, for storing a value

²The protocol does not include the refund of \mathcal{R} in case the second message is not received in time.

5 Moving Complex Computation Off-Chain

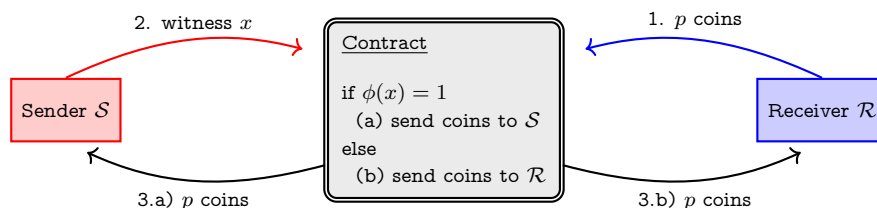


Figure 5.2: Naïve smart contract based fair sale.

x of size 1 MB in Ethereum, the parties would need to pay more than 250 euros in transaction fees (cf. gas price estimate in Section 3.2.1). These costs have to be reduced drastically if fair digital sales over the blockchain should be used in practice.

Fair sale using zero-knowledge. One appealing solution to the above problem called Zero-Knowledge Contingent Payment (ZKCP) has been proposed in [185]. ZKCP protocols use *zero-knowledge proof systems* [89] that allow a prover to convince a verifier that a specific statement is correct. In this case, the seller (in the role of the prover) uses this technique to prove to the buyer (acting as the verifier) that $\phi(x) = 1$ holds without revealing the witness x . More precisely, a ZKCP protocol between \mathcal{S} and \mathcal{R} works as follows: First, \mathcal{S} encrypts x with key k and computes a commitment $(c, d) = \text{Commit}(k)$. Moreover, he produces a zero-knowledge proof π , showing that computation of the ciphertext and the commitment was indeed done with a witness x , which satisfies $\phi(x) = 1$. Next, \mathcal{S} sends the ciphertext, the commitment c , and the zero-knowledge proof π to \mathcal{R} , who verifies the correctness of the zero-knowledge proof, and deploys a smart contract funded with p coins. It remains for the contract to wait until \mathcal{S} publishes the key k , such that the commitment can be opened correctly, i.e., $\text{Open}(c, d, k) = 1$. [185] showed that, if the underlying cryptographic primitives are secure, then the ZKCP smart contract scheme realizes a fair exchange protocol.

The execution costs of the smart contract from the ZKCP protocol are quite cheap, as it only requires the contract to evaluate commitment on a short input (the key). However, using the ZKCP puts a significant computational burden on the sender \mathcal{S} . Indeed, despite impressive progress on developing efficient zero-knowledge proof system over the last few years, current state-of-the-art schemes [53, 21, 157, 87, 187, 137] are still rather inefficient if either ϕ gets complex, or the witness x becomes large.

Non-interactive Zero-Knowledge Proofs (NIZKs) are a special kind of zero knowl-

5 Moving Complex Computation Off-Chain

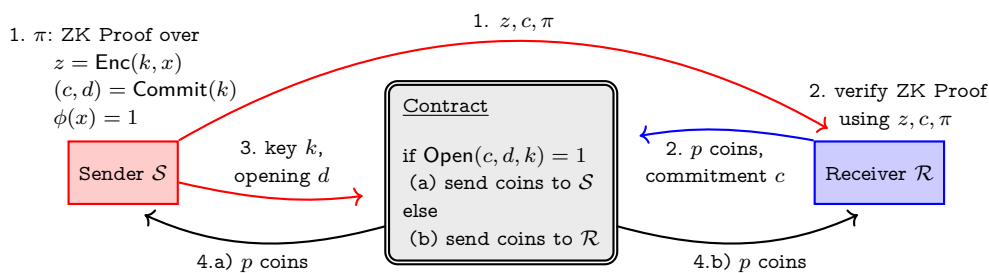


Figure 5.3: Zero-Knowledge Contingent Payment (ZKCP) based fair sale.

edge proofs that can be computed by the prover without having to interact with the sender. An additional challenge that occurs especially for NIZKs is that they require a trusted setup. In particular, a trusted entity needs to generate a Common Reference String (CRS). Zero-knowledge schemes guarantee soundness (i.e., that only true statements can be proven) and *zero knowledge*, meaning that nothing about the witness is leaked to the buyer. But if the setup is run by either \mathcal{R} or \mathcal{S} , these properties cannot be guaranteed. In fact the authors of [42] show that the ZKCP implementation of Maxwell [140] leaks information if \mathcal{R} computes the CRS. In the Sudoku example, \mathcal{R} can force \mathcal{S} to send a proof which reveals if a certain field contains some value z . In the same paper, the authors propose a solution using witness indistinguishability [42]. However, a later paper [81] showed that their protocol could again be attacked through a weak CRS.

To prevent both the heavy computation for \mathcal{S} and the difficulty of trusted setups, we aim to build a protocol without zero-knowledge proof schemes that is still efficient.

Simplified FairSwap protocol. Our construction is based on the following observations. While it is very costly (for large circuits ϕ and witnesses x) to prove that \mathcal{S} behaved correctly, it is much cheaper to prove that \mathcal{S} cheated. We show how to construct a *proof of misbehavior* which is small in size, and whose verification only requires a low number of cryptographic operations. Therefore, we can build a smart contract that can efficiently judge if the sender cheated. Only if the receiver cannot produce such a proof, \mathcal{S} will receive the locked coins from the contract.

We will now present a high-level description of the FAIRSWAP protocol (depicted in Figure 5.4), where we omit some details that we add later in the full protocol description in Section 5.4. The sender starts by running an encoding algorithm on the transcript of evaluating $\phi(x)$. In particular, this means \mathcal{S} runs the evaluation

5 Moving Complex Computation Off-Chain

of $\phi(x)$ once and encodes every intermediary step of the process. The resulting overall encoding z contains the witness x and the intermediate result of each step of ϕ . \mathcal{S} then sends z to \mathcal{R} (step 1), who cannot extract x or any information about it from the encoding without knowing the encoding key k . Once \mathcal{R} received the encoding, he sends his coins to the contract (Step 2), which prompts \mathcal{S} to reveal the key k (Step 3). Once the key is published, the receiver learns x from z . He uses an extraction function $\text{Extract}^{\mathcal{H}}$, which we will specify in more detail later. If the witness is correct, the function outputs x , \mathcal{R} sends a confirmation to the contract (step 4a), and \mathcal{S} gets his payment. However, if $\phi(x) \neq 1$, the extraction function outputs a proof of misbehavior π (step 4b), which allows \mathcal{R} to prove to the contract that he received a false witness.

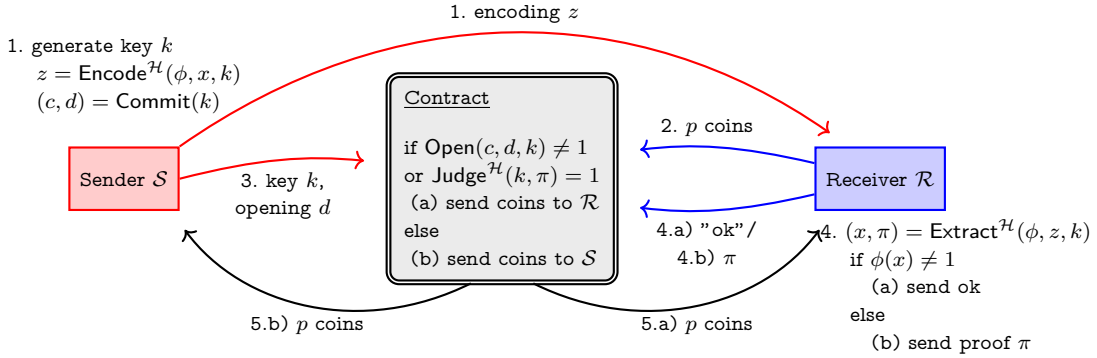


Figure 5.4: The FAIRSWAP protocol for fair sale

In order to keep the optimistic case cheap, we need to make sure that the contract does not get large inputs and does not run heavy computation. In particular, this requires keeping the value k , all public parameters in the contract small and independent of the sizes of ϕ and x . Additionally, we want to keep the pessimistic case cheap as well. We have to design secure functions $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$, and $\text{Judge}^{\mathcal{H}}$, such that the proof π is small, and the contract evaluation stays cheap. At a technical level, we use a similar technique as proposed initially in the context of multi-server delegation of computation [47], where correctness of a computation can be showed as long as a single party is honest. We need to address several technical challenges to apply this idea in our setting. In particular, our construction is non-interactive and involves only two parties. Moreover, we additionally have to provide privacy guarantees, that the witness stays hidden until the receiver has committed the coins (step 2).

Our Contribution

The main goal of FAIRSWAP is a smart contract-based protocol, which

1. achieves secure fair exchange of a witness against payment,
2. uses simple smart contracts that can be executed with low fees, and
3. avoids the use of heavy zero-knowledge proof systems to lower the computational burden on the players and prevent difficult trusted setups (like CRS).

Our protocol works for arbitrary predicate functions ϕ and witnesses x of large size. Concretely, we model ϕ as a circuit with m gates taking as input a witness $x = (x_1, \dots, x_n)$, where each x_i is represented as a bit string of length λ . We require that the gates of the circuit represent operations from some set of allowed instructions Γ . The main distinctive feature of our construction is its efficiency. Concretely, for a circuit of size m , our smart contract has asymptotic complexity of $\mathcal{O}(\log(m))$, where the hidden constants in the asymptotic notation are small. For our file sale example, the verification in the contract can be carried out using only $\mathcal{O}(\log(n))$ hash function calls. We show that running the protocol only requires a constant low amount of fees in the optimistic case (0,56 euro) and the fees for judging a dispute are low (around 1.07 euro) and only slowly increase logarithmically in the sizes of x and ϕ . Additionally, the local computation that \mathcal{R} and \mathcal{S} run is very efficient compared to zero-knowledge proofs.

Encoding scheme. We present an efficient encoding scheme that allows \mathcal{S} to send a hidden commitment of x and the step-by-step evaluation of $\phi(x)$ to the receiver. Similar to commitment schemes, this commitment cannot be changed by the sender after the fact (binding) and ensures that the receiver cannot learn any information without knowledge of the encoding key k (hiding). While k serves as an opening value for that commitment, we show how to keep this value small. In particular, we construct an encryption-like encoding scheme, whose security properties are analyzed in the Random Oracle Model (ROM).

Definitions and security analysis. The second contribution of our work is to provide a formalization of contract-based or coin aided fair exchange protocols. To this end, we follow the UC framework of Canetti [43] and develop a new ideal functionality that formally captures the security properties one would expect from a fair exchange protocol (cf. Section 5.3). In addition to providing a formal model,

we also carry out a full security analysis of our construction in the global random oracle model [46] in Section 5.5.

Implementation. Besides our conceptual contributions, we also provide a proof-of-concept implementation of our contract (cf. github.com/1EthDev/FairSwap). For our implementation, we consider the file sale example mentioned above and discuss the advantages of contracts specialized for this application in comparison to the general construction. Additionally, we benchmark the costs of deploying this contract and running our protocol over Ethereum. For more information on the details of the implementation, we refer the reader to Section 5.6.

Extensions. We discuss several extensions in Section 5.7. As a first extension, we analyze how to integrate penalties into our protocol to mitigate the risk of denial of service attacks by the sender. This is realized by also letting the sender \mathcal{S} lock q coins into the contract, which will go to the receiver \mathcal{R} if \mathcal{S} is caught cheating. Such financial penalties allow us to deal with the costs and fees for \mathcal{R} , which naturally occur in smart contract-based protocols. Concretely, we want compensation for \mathcal{R} when \mathcal{S} misbehaves, e.g., by sending a wrong file. The penalty deposit repays \mathcal{R} for his costs of interacting with the contract (e.g., for the initial contract deployment) but also for the collateral costs of locking his p coins. This addition enforces the honest behavior of rational senders.

A second extension is based on the paper [73], which discusses how to make the dispute process in FAIRSWAP interactive. While an interactive protocol can lead to more transactions and thus increased execution costs and times in the pessimistic case, the protocol is more efficient in the optimistic case. We also discuss how an interactive FAIRSWAP can be used to provide fairness of transaction fees, which are not discussed in FAIRSWAP.

To further reduce the costs of our construction, we discuss how we can run it inside off-chain state channels (see Section 5.7), when the sender and receiver wish to execute multiple recurring fair exchanges. To illustrate this setting, consider a sender that wishes to sell t commodities to a receiver. In our original protocol from above, this use case results in t repeated executions of the FAIRSWAP protocol. In state channels, on the other hand, the parties can use the contract multiple times without requiring interaction with the blockchain, thereby significantly reducing the costs of our construction. This extension allows us to amortize the on-chain costs over multiple fair exchange executions.

5.1.2 Related Work

Fair exchange is a well-studied research problem. It has been shown that without further assumptions, fair exchange cannot be achieved without a TTP [155, 188, 90]. To circumvent this impossibility, researchers have studied weaker security models – most notably, the optimistic model in which a TTP is consulted only in case one party deviates from the expected behavior [8, 40]. One may view smart-contract based solutions as a variant of optimistic protocols, where the smart contract takes the role of the TTP. In particular [125] considers a similar use case (file sharing), but the security guarantees it achieves are very different from our work: (a) the arbiter uses a cut-and-choose approach and hence for a corrupted file the probability of not detecting a cheater is non-negligible (and in fact quite high for some cases, see citation [40] in [125]); (b) due to the cut-and-choose the workload of the arbiter is large, resulting into high fees in a smart contract setting. In contrast, our solution only has a negligible error rate, and the financial costs are small. We also stress that the cost model of an arbiter and a smart contract is very different.

As mentioned above, the Zero-Knowledge Contingent Payment (ZKCP) protocols (introduced in [185]) solve the fair exchange problem by using zero-knowledge proof systems. Their first implementation (for selling solutions of Sudoku puzzles) was presented in [140], and was subsequently broken by Campanelli et al. [42]. The weakness discovered in [42] concerns all the ZKCP protocols that use NIZKs protocols [29] where the verifier generates the CRS. The authors of [42] present a fix to this problem using a tool called *Subversion-NIZK* [20] and extend the concept of ZKCP to protocols for paying for *service* (i.e.: not only for static data). [81] showed that also subversion-NIZK proofs could be attacked with weak CRSs. ZKCP protocols for cryptocurrencies that do not support contracts or scripts in the transactions were constructed in [14]. The problem with zero-knowledge systems is that it is inefficient to prove statements over large data or complex functionality. For instance, in [102] the authors show that proving in zero-knowledge the correctness of a single evaluation of a hash function (SHA256) on a witness of 64 bytes requires 3 MB of additional data transfer between the parties.

While our original motivation is to design efficient protocols for fair exchange, we emphasize that our work also has other interesting applications in the context of cryptocurrencies. In particular, we observe that our protocol offers an efficient and low-cost construction for realizing the *claim-or-refund* functionality of [23]. Claim-or-refund is used to design fair protocols for multiparty computation and

5 Moving Complex Computation Off-Chain

works as follows. In an initial preparation phase, a receiver can deposit some coins p and a function ϕ into the contract. This preparation phase is followed by two stages. First, in the claim phase, a party can claim the reward p by publishing a witness x such that $\phi(x) = 1$. Finally, in the refund phase, the receiver can refund its p coins if nobody has claimed the reward yet. It is easy to see that the above describes the fair exchange setting, where the reward corresponds to the price paid for receiving x . Bentov and Kumaresan argue that claim-or-refund can be realized with smart contracts. However, a naive implementation will result in high fee costs when ϕ is complex, or x is large. Using our protocol claim-or-refund can be realized at significantly lower costs. There has been a large body of work on using cryptocurrencies such as Bitcoin to achieve fairness in cryptographic protocols (see, e.g., [5, 122, 121, 124, 117] and many more), which utilize the claim-or-refund functionality of [23]. As our protocol provides an efficient realization of this functionality, FAIRSWAP can be used to further reduce the on-chain complexity of these protocols.

Finally, we point out that the concept of proofs of misbehavior used in our construction is a frequently applied technique in practical smart contract-based protocols. One notable example is the TrueBit protocol, developed by Teutsch and Reitwießner [177]. The idea is to outsource the potentially resource-intensive process of finding solutions for complicated computational puzzles. The system consists of *provers*, *verifiers* and *judges*, where the provers are paid for solving computationally hard tasks. As provers could lie about their results and still claim the money, they are punished whenever misbehavior is detected. The verifiers are responsible for reporting such misbehavior. They are rewarded whenever they find bugs in the solution of the provers. Again, the verifiers have the ability to lie about their results to get money, which is why there exist the judges. A judge is a computationally bounded, trusted entity backed by the blockchain security and can be implemented as a smart contract. This setting is similar to ours since we also rely on the cryptocurrency and its smart contracts to judge a dispute between two parties by verifying only a single operation instead of running complex computation off-chain. The main difference is that their protocol requires all provers to publish their solutions and interact with the verifier and judge in case they are challenged. In our case, we need the verification of misbehavior to be non-interactive, and the result of the computation should stay secret to outside observers. These restrictions add additional overhead to our protocol in comparison to a simple protocol which only helps to resolve disputes. Finally, with respect to TrueBit, we point out that its current whitepaper [177] provides only very little details on how such

proofs are created at a technical level, and no formal security analysis is provided. Hence, one may view our construction of the subroutines $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$, and $\text{Judge}^{\mathcal{H}}$ described in Section 5.4 as a building block for TrueBit, and our formal security analysis as a first step in formally analyzing the TrueBit system.

In Section 5.7.5 we will also discuss two works that extend the FAIRSWAP results; In particular, Section 5.7.2 describes the Optiswap protocol [73], which implements an interactive version of FAIRSWAP. Section 5.7.3 discusses the SmartJudge result [183], which proposes a more generic method to improve deployment costs and also presents an optimized implementation of FAIRSWAP.

5.2 Preliminaries

In this section, we introduce the notation and basic building blocks needed for the FAIRSWAP protocol specification and proof. In particular, we introduce circuits formally as we model the verification ϕ as a circuit, and we present the global ideal functionalities for the ledger and the random oracle that we use in the UC modeling.

5.2.1 Modeling Circuits

In this work, we will use circuits to model arbitrary program code over an admissible instruction set Γ . A circuit ϕ is represented by a directed acyclic graph, where the edges carry values from some set X , and the nodes represent gates. We assume that gates evaluate some instruction $\text{op} : X^\ell \rightarrow X$, where $\text{op} \in \Gamma$. A gate is evaluated by taking as input up to ℓ values from X , carrying out the instruction op , and sending the result on its outgoing wire. We limit fan-in of gates to ℓ and model arbitrary fan-out by letting the output of a gate be an input to any number of other gates. A special type of gate that we consider are input gates, which have no incoming edges (i.e., in-degree 0) and model the initial input of the circuit. We will often use the notation $\phi(x)$ to represent the output of evaluating a circuit ϕ on some input x , where the evaluation is done layer-by-layer starting with the input gates. Our construction requires a concise way to fully describe the topology and the operations of a circuit ϕ . To this end, we assign to each gate g of ϕ a label represented by a tuple $\phi_i := (i, \text{op}, I_i)$. Each such tuple consists of an instruction $\text{op} : X^\ell \rightarrow X$, which denotes the instruction carried out by this gate and a unique identifier $i \in \mathbb{N}$. The identifiers are chosen in the following way: All gates in the j th layer of the circuit have identifiers that are larger than the identifiers used by

gates in layer $j - 1$. This means that the identifier of g is larger than the identifiers of all input gates to g . Finally, the last element I_i is a set of identifiers, where $I_i = \emptyset$ if g is an input gate; otherwise, I_i is defined to be the set of identifiers of the input gates to g . In the following, we will often abuse notation and sometimes use ϕ to present the circuit (e.g., when writing $\phi(x)$ for the evaluation of ϕ on input x), or to represent the tuple of labels, i.e., $\phi = (\phi_1, \dots, \phi_m)$.

It is well known that any deterministic program can be represented by a boolean circuit. In this case, we have $X = \{0, 1\}$ and $\Gamma = \{\text{AND}, \text{NOT}\}$ are the standard binary operations, where each gate has an in-degree of at most $\ell = 2$. But in most cases, Γ will contain more powerful operations that compute on larger bit strings $\{0, 1\}^\lambda$. Examples of such higher-level instructions are hash function evaluations or modulo multiplication, which are offered by higher-level programming languages³

5.2.2 The Ledger Functionality

Again we work in the global UC model described in Section 2.4, where we model the ledger as a global functionality \mathcal{L} . It stores the coins of users and smart contracts which captures the basic properties of a cryptocurrency. Concretely, we allow parties to transfer coins between each other and explicitly allow contracts to lock coins.

Global Functionality Ledger \mathcal{L}

Functionality \mathcal{L} , running with a set of parties $\mathcal{P}_1, \dots, \mathcal{P}_n$ stores the balance $p_i \in \mathbb{N}_{\geq 0}$ for every party $\mathcal{P}_i, i \in [n]$ and a partial function $L : \{0, 1\}^* \rightarrow \mathbb{N}$ which stores how much coins are locked for each session id (initially empty). It accepts queries of the following types:

Initialization Upon receiving message (`initialize`, (p_1, \dots, p_n)) from the Environment \mathcal{Z} such that $p_i \in \mathbb{N}_{\geq 0}$ for all $i \in [n]$, store this tuple.

Update Funds Upon receiving message (`update`, \mathcal{P}_i, p) with $p \geq 0$ from \mathcal{Z} set $p_i = p$ and send (`updated`, \mathcal{P}_i, p) to every entity.

Freeze Funds Upon receiving message (`freeze`, id, \mathcal{P}_i, p) from an ideal functionality of session id check if $p_i \geq p$. If this is not the case,

³In our protocol, we require that an Ethereum smart contract can evaluate the instructions. Therefore a good candidate for the instruction set are the supported operations of the EVM or Solidity.

reply with `(no – funds, \mathcal{P}_i, p)`. Otherwise set $p_i = p_i - p$, store (id, p) in L and send `(frozen, id, \mathcal{P}_i, p)` to every entity.

Unfreeze Funds Upon receiving message `(unfreeze, id, \mathcal{P}_j, p)` from an ideal functionality of session id , check if $(id, p') \in L$ with $p' \geq p$. If this check holds update (id, p') to $(id, p' - p)$, set $p_j = p_j + p$ and send `(unfrozen, id, \mathcal{P}_j, p)` to every entity.

Let us briefly describe the functionality \mathcal{L} , whose internal state is public, and consists of the balances $p_i \in \mathbb{N}$ of parties \mathcal{P}_i and a list of contract instances. For the latter, we define a partial function $L : \{0, 1\}^* \rightarrow \mathbb{N}$ that maps a contract identifier id to a number of coins that are locked for the execution of contract id . The ledger functionality offers the following interface to the parties. The environment \mathcal{Z} can update the account balance of the users via sending an `update` message to \mathcal{L} . The parties $\mathcal{P}_1, \dots, \mathcal{P}_n$ cannot directly interact with \mathcal{L} , but their balance can be updated via `freeze/unfreeze` messages sent by other ideal functionalities. If a functionality C interacts with the ledger, we will write $C^{\mathcal{L}}$ to make this relation explicit. More precisely, `freeze` transfers money from the balance of a party to a contract functionality (identified over the session identifier id), while `unfreeze` sends this money back to the user’s account. To simplify exposition, for a malicious party \mathcal{P}_i we let the simulator Sim decide how many coins are sent back to \mathcal{P}_i ’s account by sending an `unfreeze` message.⁴ To simplify the functionalities, coins are unlocked automatically in the ideal world when the timeout expires.

Recall the communication model we specified in Section 3.4. As the functionalities in this chapter model smart contracts on the blockchain, we need to model the blockchain delay of up to Δ rounds. In particular, we model that the communication to ideal functionalities is instantaneous, but the adversary has the power to delay the reaction of the functionality upon receiving the message by up to Δ rounds. In the functionality, we write *the functionality expects a message within a certain round t* to denote this delayed execution.

5.2.3 Global Random Oracle

In addition to the global ledger functionality \mathcal{L} , we will also model the hash function as a global functionality. While in practice, collision-resistant hash functions

⁴Looking ahead this is needed to simulate the case when a malicious party in the real world decides not to request a refund and thus lock its coins in the contract.

can be instantiated with constructions such as Secure Hash Algorithm 3 (SHA-3), for our security analysis, we will assume that our hash function H is modeled as a *random oracle* \mathcal{H} . This model is common for cryptographic proofs and assumes that hash queries return perfectly random values. For proving security in the UC-model, we use a random oracle ideal functionality \mathcal{H} , which (unless otherwise instructed) responds to all queries with uniformly random sampled values $r \leftarrow \{0, 1\}^\mu$ and stores all query-response pairs (q, r) in the set Q . If a query for q has been answered before, such that $(q, r) \in Q$ is stored, \mathcal{H} responds with r .

We model hash functions as a global functionality \mathcal{H} following the works of [46, 41]. In this model, every party has oracle access to a global functionality \mathcal{H} , which represents idealized hash functions. Since we require programmability in the GUC model, we follow the work of [41] and model \mathcal{H} as a restricted *programmable* and *observable* random oracle as defined below. This functionality allows the adversary (or simulator) in the name of corrupted parties to simulate and observe only queries made from their own session (denoted with session and contract identifier id).

The first feature of the random oracle is that parties can query it on some value q , by calling (query, id, q) . For simplicity we will write $r \leftarrow \mathcal{H}(q)$. The random oracle will return a uniformly sampled value or an already existing value r from the set Q . Additionally to this straightforward functionality, we require *programmability*, which is a special property needed for proving security in the UC model. Programmability means that the adversary is allowed to fix the response of the oracle for certain queries if they have not been queried before by sending $(\text{program}, q, r)$ (we say the adversary *programs* the random oracle).

It is common for UC security proofs to work with programmable and/or observable random oracles. Programmability means that the ideal UC adversary Sim can control the random oracle and program its hashes to specific responses. Additionally, it can see all queries made by the environment \mathcal{Z} to the random oracle. Traditionally, such a random oracle is modeled as a local functionality, which is in control of the simulator. This again implies that every unique protocol execution has its own local disjoint hash function. Since we want to explicitly allow the composition of multiple protocols, we follow the argument of [46, 41] that such local functionalities are not a good model for a standard hash function like SHA-3. A global functionality would respond to all queries in all sessions with the same values, which cannot be done with local random oracle functionalities. Programmable random oracles are a useful and practical tool in many UC simulations, which has been studied intensively before in the non-global setting [152, 79]. In the non-global UC model, the simulator requires these properties to sim-

5 Moving Complex Computation Off-Chain

ulate indistinguishable commitments. In the global UC model, there might be multiple executions that all interact with the same global random oracle, which gives the power to the environment to compare queries and send influence from different (parallel) executionst⁵. Intuitively, this adversarial power seems to break the security of schemes based on this functionality, since any adversary is allowed to program collisions. But \mathcal{H} ensures that this is not the case. As a protection, parties have the ability to verify if some response of the random oracle has been programmed by calling $\mathcal{H}(\text{isPrgrmd}, r)$. If \mathcal{H} responds with 1, the parties know that the value was programmed and reject it.

Global Functionality \mathcal{H} (from [41])
<p>The \mathcal{H} functionality is the global random oracle with restricted programming and observability, which takes as input queries $q \in \{0, 1\}^*$ and outputs values $r \in \{0, 1\}^\mu$. Internally it stores initially empty sets Q, P and a set Q_{id} for all sessions id.</p>
Query
<p>Upon receiving message (query, id, q) from a party of session $id' \neq id$ proceed as follows:</p> <ul style="list-style-type: none"> • If $(id, q, r) \in Q$ respond with (query, q, r). • If $(id, q, r) \notin Q$ sample $r \in \{0, 1\}^\mu$, store (id, q, r) in Q and respond with (query, q, r). • If the query is made from a different session ($id \neq id'$), store (q, r) in Q_{id}.
Program
<p>Upon receiving message $(\text{program}, id, q, r)$ by the adversary \mathcal{A} check if (id, q, r') is defined in Q. If this is the case, abort. Otherwise, if $r \in \{0, 1\}^\mu$ store (id, q, r) in Q and (id, q) in P.</p> <p>Upon receiving message $(\text{isPrgrmd}, q)$ from a party of session id check if $(id, q) \in P$. If this is the case respond with $(\text{isPrgrmd}, 1)$.</p>
Observe
<p>Upon receiving message (observe) from the adversary of session id respond with $(\text{observe}, Q_{id})$.</p>

⁵In our case, the environment could access the global random oracle through the adversary of another session or protocol execution and program collisions.

Additionally to the restricted programmability, the functionality \mathcal{H} allows leakage of all illegitimate queries, which were made by the environment over the adversary, by sending a `observe` message. The functionality \mathcal{H} will respond with the set Q_{id} , which contains all illegitimate queries made from that session. This includes all queries from adversaries that are not from the desired session. For more information about the construction and properties of this ideal functionality, we refer the reader to [41].

5.2.4 Constructing the FairSwap Building Blocks in the Random Oracle Model

We will introduce some further cryptographic building blocks in this chapter. In particular, we show how to construct Merkle trees and a commitment scheme in the previously defined global random oracle with restricted programmability and observability.

Merkle Trees

A Merkle hash tree or Merkle tree is often used to create a short hash out of to a large number of elements. The elements form the leaf nodes of the tree and two nodes are iteratively hashed together to form a single root element. The tree root serves as a digest of all elements and as the same time a short (logarithmic) proof suffices to prove that a single element is included in the tree. We consider three algorithms – $\text{Mtree}^{\mathcal{H}}$ to create the tree, $\text{Mproof}^{\mathcal{H}}$ to generate the logarithmic proof that a single element is included and $\text{Mvrfy}^{\mathcal{H}}$ to verify if such a proof is correct. Since \mathcal{H} , which is queried internally in all three algorithms, is a global random oracle with restricted programmability and observability, we construct the three algorithms below, e.g., to ensure that Merkle Proof with programmed values are not accepted.

Formally, we follow the standard notation as defined in [67]. A *Merkle tree* M of elements $x_1, \dots, x_n \in \{0, 1\}^*$ (where for simplicity n is an integer power of 2) is a labeled binary tree $M = \text{Mtree}^{\mathcal{H}}(x_1, \dots, x_n)$ with the i -th leaf is labeled by x_i (denoted as $\text{label}(x_i) = x_i$). We will denote leaf nodes and their labels with x and non-leaf nodes with V and their labels with v . Moreover, a label v_j of every non-leaf node V_j is the hash of the labels v_j^l and v_j^r of its two child nodes V_j^l and V_j^r respectively (i.e., $v_j := \mathcal{H}(v_j^l, v_j^r)$). We call V_j the **parent** of v_j^l and v_j^r . We say v_j^l is a **sibling** of v_j^r and vice versa. A Merkle tree M of n elements x_1, \dots, x_n is

5 Moving Complex Computation Off-Chain

created with the $\text{Mtree}^{\mathcal{H}}$ algorithm (cf. Algorithm 1).

Algorithm 1 Merkle tree hash $\text{Mtree}^{\mathcal{H}}$

Input: (x_1, \dots, x_n)

- 1: **set** V $\triangleright V$ will be the root node
- 2: **if** $n = 1$ **then** \triangleright If input is single value, V is a leaf
- 3: $label(V) = x_1$ \triangleright assign label of leaf V as x_1
- 4: **else** \triangleright otherwise recursively call $\text{Mtree}^{\mathcal{H}}$ algorithm again
- 5: $v_0^l = \text{Mtree}^{\mathcal{H}}(x_1, \dots, x_{\lceil n/2 \rceil})$
- 6: $v_0^r = \text{Mtree}^{\mathcal{H}}(x_{\lceil n/2 \rceil + 1}, \dots, x_n)$
- 7: $label(V) = \mathcal{H}(\text{root}(v_0^l) || \text{root}(v_0^r))$ \triangleright label = hash of subtree
- 8: Let M be a binary tree with root node label V and left subtree v_0^l and right subtree v_0^r

Output: Merkle tree M with root V

The label at the root of a Merkle tree M is denoted by $\text{root}(M)$. For efficiently proving that an element x_i is included in the Merkle tree (identified over its root hash h), we use a *Merkle proof* ρ , which is a vector (of length $\mathcal{O}(\log(n))$) consisting of labels on all the siblings of elements on a path from the i -th leaf to the root of the Merkle tree. We denote the algorithm for generating a Merkle proof by $\text{Mproof}^{\mathcal{H}}$, which on input a Merkle tree M and an index i outputs a Merkle proof ρ that x_i is the i -th leaf of M (cf. Algorithm 2).

Algorithm 2 Merkle tree proof $\text{Mproof}^{\mathcal{H}}$

Input: Merkle tree M , index i

- 1: $V = M[i]$ \triangleright let V be the i -th leaf node of M
- 2: **for each** $j \in [\log_2(n)]$ **do**
- 3: **set** $l_j = label(\text{sibling of } v)$
- 4: **set** $v = \text{parent of } v$

Output: Merkle Proof $\rho = (l_1, \dots, l_d)$

Finally, the algorithm $\text{Mvrfy}^{\mathcal{H}}$ with oracle access to \mathcal{H} takes as input an element x_i , a Merkle proof ρ and a root of a Merkle tree $\text{root}(M)$. The algorithm $\text{Mvrfy}^{\mathcal{H}}$ verifies if the i -th leaf element x corresponds to a Merkle tree with root r (generated with Algorithm 1) using proof ρ (generated with Algorithm 2). If the verification holds, the algorithm outputs 1 if the verification fails, the algorithm outputs 0.

If the root r of the tree is known beforehand, this algorithm can be used to verify that x is the i -th element of a Merkle tree with root r . As the random oracle functionality \mathcal{H} that we consider can be programmed, however, we need to ensure additional precautionary measures. In particular, we need to prevent a honest verifier accepts a proof that is forged by programming hash values. This

Algorithm 3 Merkle tree proof verification $\text{Mvrfy}^{\mathcal{H}}$

Input: $i \in [n], x \in \{0, 1\}^\lambda, \rho = (l_1, \dots, l_d), h \in \{0, 1\}^\mu$

- 1: **for each** $l_j \in \rho$ **do**
- 2: **if** $i/2^j = 0 \pmod 2$ **then**
- 3: $x = \mathcal{H}(l_k || x)$
- 4: **if** $\mathcal{H}(\text{isPrgrmd}(l_k || x))$ **then**
- 5: **Terminate and Output** \perp \triangleright reject hash is programmed
- 6: **else**
- 7: $x = \mathcal{H}(x || l_j)$
- 8: **if** $\mathcal{H}(\text{isPrgrmd}(x || l_j))$ **then**
- 9: **Terminate and Output** \perp \triangleright reject hash is programmed
- 10: **if** $x = h$ **then**
- 11: **Output** 1
- 12: **else**
- 13: **Output** 0

can be prevented easily however, as the isPrgrmd query reveals exactly these information to honest parties. Therefore every hash value in the path needs to be checked as part of the $\text{Mvrfy}^{\mathcal{H}}$ algorithm and the algorithms outputs 0 if any programmed value is encountered.

Extending Merkle Trees to Commitments For our protocol Π_{FAIRSWAP} , we need that both parties \mathcal{S} and \mathcal{R} jointly commit to the values x using a Merkle tree commitment towards the smart contract. The commitment on the values $x = (x_1, \dots, x_n)$ is generated using randomly sampled $d = (d_1, \dots, d_n)$, $d_i \in \{0, 1\}^\kappa$ as follows:

$$\begin{aligned} \text{let } x' &= (x_1 || d_1, \dots, x_n || d_n) \\ \text{Commit}(x) &= (\text{root}(\text{Mtree}^{\mathcal{H}}(x')), d) = (c, d) \\ \text{Open}(c, x, d) &= \begin{cases} x, & \text{if } \text{root}(\text{Mtree}^{\mathcal{H}}(x')) = c \\ 0, & \text{otherwise} \end{cases} \end{aligned}$$

The scheme is hiding, as long as the randomness $d \in \{0, 1\}^\kappa$ is chosen uniformly at random in the ro because then the commitments are indistinguishable for any PPT adversary. Note, that this commitment scheme does not satisfy the binding property if the random oracle is programmable since the adversary has the power to program two values x, y to result in the same response $\mathcal{H}(x) = \mathcal{H}(y)$. In our protocol, we do not need classical hiding when at least one of the parties \mathcal{S} or \mathcal{R} is honest (which is the only case in which our protocol satisfies fairness). In our case,

\mathcal{S} generates the commitment (which he will do in the correct way if he is honest) and sends it together with the committed values to \mathcal{R} . The receiver \mathcal{R} recomputes the commitment and additionally checks if any of the labels of the Merkle tree are programmed in \mathcal{H} . If he encounters any programmed value, an honest \mathcal{R} will reject the root r and abort the protocol execution. This ensures that (as long as there is one honest party), the commitment is binding.

5.2.5 Commitment Scheme Construction

Next, we construct the commitment scheme in the programmable random oracle model. Note, that it is not easily possible to use UC-style commitment functionality here since our smart contract hybrid functionality needs to run the open procedure. In the UC-model functionalities are permitted from interacting with other functionalities, this prevents us from using ideal commitment functionalities.

Let κ be the security parameter and $(a||b)$ denote the concatenation of two values a and b . Then we construct a commitment scheme (**Commit**, **Open**) in the global programmable random oracle model as described in Algorithm 4.

Algorithm 4 Algorithm Commit

Input: $x \in \{0, 1\}^*$

- 1: $d \leftarrow \{0, 1\}^\kappa$ s.t. $\mathcal{H}(\text{isPrgrmd}, x||d) \neq 1$ ▷ choose d uniformly at random
- 2: $c \leftarrow \mathcal{H}(x||d)$ ▷ query the oracle on $x||d$

Output: (c, d)

To show that this scheme is hiding, it needs to hold that any PPT algorithm \mathcal{A} cannot distinguish two commitments. From the randomness of the outputs of \mathcal{H} it follows that this construction is hiding because the output of $\mathcal{H}(x) \approx_c \mathcal{H}(y)$ is indistinguishable if the \mathcal{A} does not know (or programmed) $\mathcal{H}(x)$ or $\mathcal{H}(y)$ (which by chance only happens with a negligible probability for computationally bounded distinguishers). If d is chosen uniformly at random from domain $\{0, 1\}^\kappa$ and κ large enough the probability of guessing d is computationally hard, which means \mathcal{A} cannot distinguish **Commit**(x) from **Commit**(y) except with negligible probability.

In order to break the binding property, an adversary \mathcal{A} needs to find a collision $\mathcal{H}(x) = \mathcal{H}(y)$, without programming \mathcal{H} . Since the outputs of \mathcal{H} are uniformly distributed, the best strategy for \mathcal{A} is to guess values and query \mathcal{H} on them. If μ is large, this is hard for computationally bounded adversaries, since they can only make a polynomial in κ number of queries to \mathcal{H} . Thus, the scheme is computationally binding.

Algorithm 5 Algorithm Open

Input: $c \in \{0, 1\}^\mu, d \in \{0, 1\}^\kappa$

- 1: $c' \leftarrow \mathcal{H}(x||d)$ ▷ query the oracle on $x||d$
- 2: **if** $c == c'$ **and** $\mathcal{H}(\text{isPrgrmd}, x||d) \neq 1$ **then**
- 3: $b = 1$ ▷ ensure that the commitment was not programmed
- 4: **else**
- 5: $b = 0$ ▷ otherwise reject the opening

Output: b

5.3 Ideal Functionality for Coin aided Fair Exchange

Our ideal functionality $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ describes a setting where \mathcal{S} sells a witness x to a receiver \mathcal{R} and obtains p coins if this witness was correct. The correctness of the witness is defined through a predicate function ϕ , which for a valid input x outputs 1, and 0 otherwise. Internally, $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ will interact with the global ledger functionality \mathcal{L} to maintain the balance of the parties during the fair exchange (for instance, when a witness was successfully sold, then p coins are unfrozen in \mathcal{S} 's favor).

Functionality $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ for coin aided fair exchange
The ideal functionality $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ (in session id) interacts with a receiver \mathcal{R} , a sender \mathcal{S} , the ideal adversary Sim and the global ledger \mathcal{L} .
Initialize
<p>(Round 1) Upon receiving $(\text{sell}, id, \phi, p, x)$ with $p \in \mathbb{N}$ from \mathcal{S}, leak $(\text{sell}, id, \phi, p, \mathcal{S})$ to Sim, store witness x, circuit ϕ and price p.</p> <p>(Round 2) Upon receiving $(\text{buy}, id, \phi, p)$ from receiver \mathcal{R} in the next round, leak $(\text{buy}, id, \mathcal{R})$ to Sim and send $(\text{freeze}, id, \mathcal{R}, p)$ to \mathcal{L}. If \mathcal{L} responds with $(\text{frozen}, id, \mathcal{R}, p)$ go to Reveal phase.</p>
Reveal
<p>(Round 3) Upon receiving (abort, id) from Sim taking the role of the corrupted sender \mathcal{S}^* in round 3, send $(\text{unfreeze}, id, p, \mathcal{R})$ to \mathcal{L} in the next round and terminate. Otherwise if no such message was received in round 3, then send (bought, id, x) to \mathcal{R} and go to Payout phase.</p>
Payout

(Round 4) Upon receiving (abort, id) from Sim taking the role of the corrupted receiver \mathcal{R}^* , wait until round 5 to send (sold, id) to \mathcal{S} , $(\text{unfreeze}, id, p, \mathcal{S})$ to \mathcal{L} and terminate. Otherwise, if no such message was received:

- If $\phi(x) = 1$, send messages $(\text{unfreeze}, id, p, \mathcal{S})$ to \mathcal{L} and (sold, id) to \mathcal{S} ,
- If $\phi(x) \neq 1$, send messages $(\text{unfreeze}, id, p, \mathcal{R})$ to \mathcal{L} and $(\text{not sold}, id)$ to \mathcal{S} .

The functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ has three phases, which we first describe for the case when the parties are honest. During the *initialization phase* the ideal functionality receives inputs from both \mathcal{S} and \mathcal{R} . \mathcal{S} sends the input x and a description of the predicate circuit ϕ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. If \mathcal{R} confirms this request, the functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ instructs \mathcal{L} to freeze p coins from \mathcal{R} . If this is not possible due to insufficient funds, the functionality ends the fair exchange protocol. During the *reveal phase*, the receiver will learn x , after which the *payout phase* is started. In the payout phase, we consider two cases. If $\phi(x) = 1$, then the sender \mathcal{S} receives the coins as a payment; otherwise if $\phi(x) \neq 1$, the functionality instructs \mathcal{L} to send the coins back to \mathcal{R} .

In addition to the above, malicious parties can abort the execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in both the *reveal phase* and the *payout phase*. Concretely, during the *reveal phase*, a malicious sender \mathcal{S}^* may abort, which means the funds are sent back to \mathcal{R} . On the other hand, a malicious receiver \mathcal{R}^* may abort the exchange during the *payout phase*, which means \mathcal{S} receives the coins. Both of these cases account for the fact that in the protocol execution, a malicious party may abort and does not send the required message. Looking ahead, in the protocol a malicious sender \mathcal{S}^* may not reveal the key to the contract, leads to a refund of the locked coins to \mathcal{R} . On the other hand, a malicious receiver \mathcal{R}^* may not complain during the payout phase, even though he received a witness x with $\phi(x) \neq 1$. In this case, the funds must go to \mathcal{S} because from the contract's point of view, the case when a malicious receiver did not complain (even though $\phi(x) \neq 1$) is indistinguishable from the case when $\phi(x) = 1$.

Informal Security Properties. Let us now discuss what security properties are guaranteed by our ideal functionality. Since our protocol realizes the ideal functionality, these security properties are also achieved by our protocol in the real

world.

Termination. If at least one party is honest, the fair exchange protocol terminates within at most 5 steps (5δ rounds), and all coins are unlocked from the contract.

Sender Fairness. An honest sender \mathcal{S} is guaranteed that the receiver \mathcal{R} only learns the witness iff he pays p coins to \mathcal{S} .

Receiver Fairness. An honest receiver \mathcal{R} is ensured that he only pays p coins iff the sender delivers the correct witness in exchange.

Let us take a closer look at why these properties are realized by the ideal functionality. In the purely honest case, it is trivial to see that all properties hold. Now assume the case of a malicious receiver \mathcal{R}^* instead. The ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ only proceeds to the `reveal` phase if the receiver has locked p coins into the contract during initialization. Then, in the `payout` phase these coins are only given to \mathcal{R}^* iff $\phi(x) \neq 1$. In all other cases (i.e., if $\phi(x) = 1$ or a malicious \mathcal{R}^* aborts), \mathcal{S} receives p coins as required by sender fairness.

Finally, we consider a malicious sender \mathcal{S}^* , who only receives a payment during the payout phase if either $\phi(x) = 1$ (i.e., the witness was valid), or the receiver aborts in Step (4*), which an honest receiver never would do. This implies receiver fairness. Conclusively, it is easy to see that the ideal functionality will terminate after at most 5 steps, which may happen during payout when a malicious receiver \mathcal{R} aborts.⁶

5.4 FairSwap Protocol

As highlighted in the overview section, we will solve the disagreement where the sender \mathcal{S} claims that he sent a witness x such that $\phi(x) = 1$ to the receiver \mathcal{R} , while \mathcal{R} claims the contrary. To resolve this conflict, we will use a smart contract to act as a *judge* and can decide which of both cases occurred. In order to minimize costs for the execution of this contract, we do not want the judge contract to learn ϕ , x , nor require it to run $\phi(x)$. Instead, we outsource the heavy work of evaluating the circuit to \mathcal{S} and \mathcal{R} , respectively. The judge contract will only need to verify a *concise proof of misbehavior*, which \mathcal{R} generates if he wants to complain about

⁶Note, the ideal functionality does not provide any fairness or termination guarantees for two corrupted parties.

5 Moving Complex Computation Off-Chain

the fact that $\phi(x) \neq 1$. We will show in this section how to generate such a proof, whose size is logarithmic in the circuit size representing ϕ . This is an important property, since we allow the witness x and therefore also ϕ to be large, i.e., x may consist of n elements, $x = (x_1 \dots, x_n)$, with $x_i \in \{0, 1\}^\lambda$. The circuit ϕ takes as input x and has $m > n$ gates, which are evaluated according to the topology of the circuit where gate g_m is the output gate of the overall circuit. If the operation of gate g_m outputs 1, the circuit ϕ accepts the witness x . Otherwise, the witness is rejected.

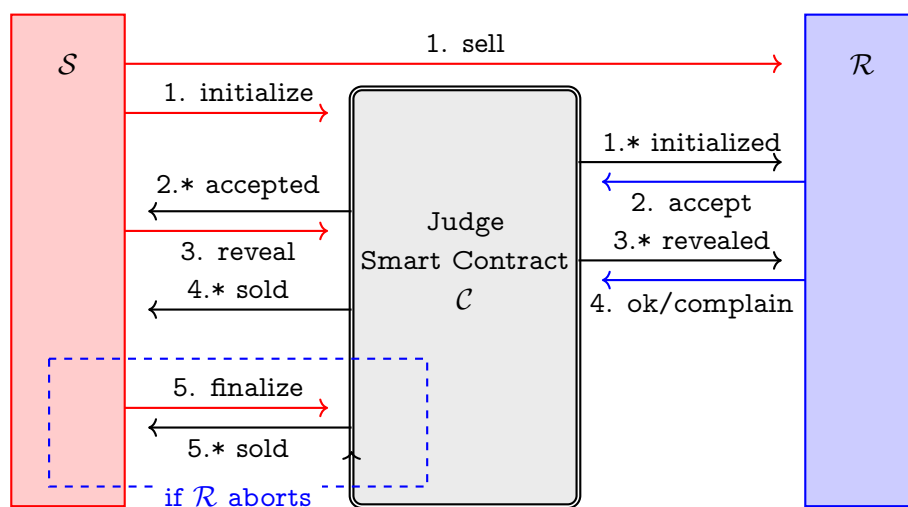


Figure 5.5: Outline of fair exchange with judge contract

We propose a new scheme that, at a high-level, works as follows (cf. also Figure 5.5). In the first step, the sender \mathcal{S} encodes x and auxiliary information about the computation of $\phi(x)$ and sends these ciphertexts to the receiver \mathcal{R} (Step 1a). In the same step, it sends a commitment of the key k used for the encoding to the judge contract (Step 1b). The receiver does some preliminary consistency checks in the next step and (if he accepts) sends p coins to the judge contract (Step 2). In the third step, the sender is supposed to reveal the key k to the judge contract (Step 3). This enables \mathcal{R} to extract x and verify the computation of $\phi(x)$. If x was not correct, i.e., $\phi(x) \neq 1$, then \mathcal{R} has the chance to complain about the invalid x in the fourth step via a concise proof of misbehavior (Step 4). In this case, the p coins locked by \mathcal{R} in the contract get refunded. Finally, in case \mathcal{R} was malicious and did not react in step 4, \mathcal{S} can finalize the smart contract in the fifth step.

5.4.1 Witness and Transcript Encoding Scheme

Before we describe our main protocol in detail, we start by taking a closer look at how \mathcal{S} generates the encoding z , and how \mathcal{R} can either extract the witness x or generate the complaint for the judge smart contract. We will consider three algorithms, the $\text{Encode}^{\mathcal{H}}$ algorithm run by \mathcal{S} , the $\text{Extract}^{\mathcal{H}}$ algorithm run by \mathcal{R} , and the $\text{Judge}^{\mathcal{H}}$ algorithm run by the smart contract. \mathcal{S} takes x as an input and outputs an encoding z , which hides x from the receiver before the encoding key k is published. $\text{Extract}^{\mathcal{H}}$ will take z and k and extract either the witness x (if $\phi(x) = 1$) or output a proof of misbehavior π . If $\text{Judge}^{\mathcal{H}}$ gets such a proof π over an incorrect witness x , s.t. $\phi(x) \neq 1$ the algorithm outputs 1 and 0 otherwise. We present our protocol in a modular way using the subroutines $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$, and $\text{Judge}^{\mathcal{H}}$ shown in Algorithms 1-3.

The challenges for this construction are (i) to keep all data that is sent to the contract small and (ii) reduce the computation inside the contract. The key idea is to let the judge contract check that \mathcal{S} carried out some part of the claimed computation incorrectly instead of verifying the correctness of the entire computation. In our construction, we let the judge validate only the operation and the result of a single incorrectly computed gate of ϕ . This is done via a *concise proof of misbehavior*. Such a proof includes the inputs $\pi_{in}^1, \dots, \pi_{in}^\ell$, and the output out_i of the gate ϕ_i where $\phi_i = (i, \text{op}_i, I_i)$ specifies the index i within the circuit, the operation op_i and the set of indices of the input wires I_j . Thus we need to ensure that the judge gets all these inputs as part of the proof. In particular, we need to prevent \mathcal{R} from sending values for different indices so the judge contract can be ensured that the values used for the proof of misbehavior were originally generated by \mathcal{S} . We ensure this by an efficient commitment using Merkle trees.

Encode Algorithm. In the FAIRSWAP protocol, \mathcal{S} uses the algorithm $\text{Extract}^{\mathcal{H}}$ to encode x and the intermediate values that are produced during the evaluation of $\phi(x)$ (cf., Algorithm 1). The output z of this encoding procedure is sent to the receiver \mathcal{R} . Moreover, as described above, \mathcal{S} sends a commitment of the key k to the smart contract.

The encoding of every element is done by hashing the key k together with the index of the gate i , which results in a new random string of the length μ (as given by the random oracle).

$$z_i = x_i \oplus \mathcal{H}(k||i)$$

As all elements x_i also have at most this length, the bitwise xor leads to a random

Algorithm 6 Encode ^{\mathcal{H}} (ϕ, x)

Input: $\phi = (\phi_1, \dots, \phi_m)$ and $x = (x_1, \dots, x_n)$

- 1: $k \leftarrow \{0, 1\}^\kappa$ s.t. $\forall i \in [m] : \mathcal{H}(\text{isPrgrmd}(k||i)) \neq 1$ \triangleright sample k
- 2: **for each** $i \in [n]$ **do**
- 3: $\text{out}_i = x_i$
- 4: $k_i = \mathcal{H}(k||i)$ \triangleright generate i -th key
- 5: $z_i = k_i \oplus x_i$ \triangleright Encode witness through xor with i -th key
- 6: **for each** $i \in \{n + 1, \dots, m\}$ **do**
- 7: **parse** $\phi_i = (i, \text{op}_i, I_i)$
- 8: $\text{out}_i = \text{op}_i(\text{out}_{I_i[1]}, \dots, \text{out}_{I_i[\ell]})$ \triangleright Compute the i -th operation
- 9: $k_i = \mathcal{H}(k||i)$ \triangleright generate i -th key
- 10: $z_i = k_i \oplus \text{out}_i$ \triangleright Encode output values through xor with i -th key

Output: $z = (z_1, \dots, z_m), k$

ciphertext, similar to the computation of a one-time pad. One may say we use the hash function to extend the domain of the key or generate a counter mode style encryption of the elements⁷ As we work in the programmable random oracle, we avoid using keys that have been programmed before the encoding. We ensure this by selecting a key randomly and checking if it has been programmed for any of the indices 1 to m .

Extract Algorithm. Once \mathcal{S} reveals the encoding key k , \mathcal{R} can run the extraction subroutine Extract ^{\mathcal{H}} (cf. Algorithm 2) and recover x . The algorithm gets as input the encoding z , the circuit ϕ , the key k and outputs a tuple, where the first element is the decoding of the witness x and the second is either \perp (if $\phi(x) = 1$) or a concise proof of misbehavior π (if $\phi(x) \neq 1$). The proof π is used later to convince the judge/contract that some step of the computation of $\phi(x)$ is incorrect.

On input the decoding key k , the root elements r_z and r_ϕ , and the proof π the algorithm Judge ^{\mathcal{H}} outputs 1 if the complaint succeeds or 0 otherwise (cf. Algorithm 3). In order to verify the i -th step of $\phi(x)$, the judge needs to know the label $\phi_i = (\text{op}_i, i, I_i)$, all values $\text{out}_{I_i[1]}, \dots, \text{out}_{I_i[\ell]}$ on its input wires and the value of its output wire out_i . Using this information, the algorithm computes the output of the i -th gate and compares it with the value out_i . If both values are the same, then the computation was carried out correctly, and the algorithm outputs 0 (i.e., it rejects the complaint). Otherwise, it outputs 1, and we say that the judge

⁷We note that this is not an encryption scheme as it only works for one-time encryption, as we reveal the key and we require a stronger property from commitments, i.e., the binding property.

5 Moving Complex Computation Off-Chain

algorithm *accepts* the complaint.

Algorithm 7 Extract $^{\mathcal{H}}(\phi, z, k)$

Input: $\phi = (\phi_1, \dots, \phi_m), z = (z_1, \dots, z_n), k$

- 1: **for each** $i \in [n]$ **do**
- 2: $k_i = \mathcal{H}(k||i)$ ▷ Generate i -th key
- 3: $x_i = k_i \oplus z_i$ ▷ Extract witness by xor of key and encoding
- 4: $M_z = \text{Mtree}^{\mathcal{H}}(z)$ ▷ Compute Merkle tree over z
- 5: $M_\phi = \text{Mtree}^{\mathcal{H}}(\phi)$ ▷ Compute Merkle tree over ϕ
- 6: **if** $\exists i \in [n]$ s.t. $\mathcal{H}(\text{isPrgrmd}(k||i))$ **then**
- 7: $\pi_\phi = (\phi_i, \text{Mproof}^{\mathcal{H}}(i, M_\phi))$ ▷ Proof that $\phi_i \in \phi$
- 8: $\pi_{\text{out}} = (z_i, \text{Mproof}^{\mathcal{H}}(i, M_z))$ ▷ Proof that $z_i \in z$
- 9: **set** $\pi = (\pi_\phi, \pi_{\text{out}}, \emptyset)$
- 10: **Terminate and Output:** $((x_1, \dots, x_n), \pi)$ ▷ Output complaint if any key is programmed
- 11: **for each** $i \in \{n+1, \dots, m\}$ **do**
- 12: **parse** $\phi_i = (i, \text{op}_i, I_i)$
- 13: $\text{out}_i = \text{op}_i(\text{out}_{I_i[1]}, \dots, \text{out}_{I_i[\ell]})$ ▷ Compute output of i -th gate
- 14: $k_i = \mathcal{H}(k||i)$ ▷ Generate i -th key
- 15: $\text{out}'_i = k_i \oplus z_i$ ▷ Extract output by xor of key and encoding
- 16: **if** $\text{out}'_i \neq \text{out}_i$ or $(i = m \text{ and } \text{out}_i \neq 1)$ or $(\mathcal{H}(\text{isPrgrmd}(k||i)))$ **then**
- 17: $\pi_\phi = (\phi_i, \text{Mproof}^{\mathcal{H}}(i, M_\phi))$ ▷ Proof that $\phi_i \in \phi$
- 18: $\pi_{\text{out}} = (z_i, \text{Mproof}^{\mathcal{H}}(i, M_z))$ ▷ Proof that $z_i \in z$
- 19: **for each** $k \in [\ell]$ **do**
- 20: **set** $j = I_i[k]$ ▷ j is the k -th index in set I_i
- 21: $\pi_{in}^k = \text{Mproof}^{\mathcal{H}}(j, M_z)$ ▷ Proof that $z_j \in z$
- 22: **set** $\pi = (\pi_\phi, \pi_{\text{out}}, \pi_{in}^1, \dots, \pi_{in}^\ell)$
- 23: **Output:** $((x_1, \dots, x_n), \pi)$
- 24: **Output:** $((x_1, \dots, x_n), \perp)$

In particular if the algorithm encounters any programmed hash value, it also accepts the complaint. This is important to prevent that the sender programs the global random oracle \mathcal{H} to make a false file hash to the correct root hash. This attack is captured by also generating a proof of misbehavior and sending it to the judge. Now it remains to construct the judge algorithm that correctly verifies the complaint.

Judge Algorithm. To guarantee that \mathcal{R} can only complain about values that he has indeed received from \mathcal{S} and that violate the predicate function ϕ on which both \mathcal{S} and \mathcal{R} have agreed on, we require that the Merkle roots $r_z = \text{root}(\text{Mtree}^{\mathcal{H}}(z))$ and $r_\phi = \text{root}(\text{Mtree}^{\mathcal{H}}(\phi))$ are stored in the judge contract. Concretely, \mathcal{S} sends r_z

5 Moving Complex Computation Off-Chain

and r_ϕ to the contract in the first step, and \mathcal{R} will only deposit p coins into the contract if these values are consistent with z . When later $\text{Judge}^{\mathcal{H}}$ receives a concise proof of misbehavior $\text{Judge}^{\mathcal{H}}$ checks if the containing Merkle proofs are consistent with r_z and r_ϕ . Only if this is the case, a complaint is accepted by the contract.

Algorithm 8 $\text{Judge}^{\mathcal{H}}(k, r_z, r_\phi, \pi)$

```

1: parse  $\pi = (\pi_\phi, \pi_{\text{out}}, \pi_{in}^1, \dots, \pi_{in}^\ell)$ 
2: parse  $\pi_\phi = (\phi_i, \rho_\phi)$ 
3: parse  $\phi_i = (i, \text{op}_i, I_i)$  ▷ Reject if  $\phi_i$  not  $i$ -th step of  $\phi(x)$ 
4:  $k_i = \mathcal{H}(k||i)$  ▷ Generate  $i$ -th key
5: if  $\mathcal{H}(\text{isPrgrmd}(k||i))$  then
6:   Terminate and Output 1 ▷ Accept if any key is programmed
7: else
8:    $\text{out}_i = k_i \oplus z_i$  ▷ Extract output by xor of key and encoding
9: if  $\text{Mvrfy}^{\mathcal{H}}(\phi_i, \rho_\phi, r_\phi) \neq 1$  output: 0
10: parse  $\pi_{\text{out}} = (z_i, \rho_{\text{out}})$  ▷ Reject if  $z_i$  not  $i$ -th element of  $z$ 
11: if  $\text{Mvrfy}^{\mathcal{H}}(z_i, \rho_{\text{out}}, r_z) \neq 1$  output: 0
12: if  $i = m$  and  $\text{out}_i \neq 1$  output: 1 ▷ Accept if  $\phi(x) \neq 1$ 
13: for each  $j \in [l]$  do ▷  $j$  is the  $k$ -th index in set  $I$ 
14:   parse  $\pi_{in}^j = (z_j, \rho_j)$  ▷ Reject if  $z_j$  not  $z[j]$ 
15:   if  $\text{Mvrfy}^{\mathcal{H}}(z_j, \rho_j, r_z) \neq 1$  output: 0
16:    $k_{I_i[j]} = \mathcal{H}(k||I_i[j])$  ▷ Generate  $I_i[j]$ -th key
17:   if  $\mathcal{H}(\text{isPrgrmd}(k||I_i[j]))$  then
18:     Terminate and Output 1 ▷ Accept if any key is programmed
19:   else
20:      $\text{out}_{I_i[j]} = k_{I_i[j]} \oplus z_j$  ▷ Extract output by xor of key and encoding
21: if  $\text{op}_i(\text{out}_{I_i[1]}, \dots, \text{out}_{I_i[l]}) \neq \text{out}_i$  output: 1 ▷ Accept
22: Else Output: 0 ▷ Reject complaint if evaluation correct

```

This concise proof of misbehavior π consists of a total of $\ell + 2$ Merkle proofs, and hence the complexity of the judge is $O(\ell \log(m))$. The first element $\pi_\phi \in \pi$ includes the Merkle proof that shows that label ϕ_i is indeed the label corresponding to the i -th gate in ϕ . The second element π_{out} includes a Merkle proof ρ_{out} , which is required to verify that z_i is indeed the i -th element in z . Finally, π contains Merkle proof $\pi_{in}^1, \dots, \pi_{in}^\ell$ for the ℓ encoded input values of the gate with label ϕ_i . Given these Merkle proofs, the judge algorithm verifies their correctness, extracts z_i of the i -th operation ϕ_i into the output value out_i . Then, it checks whether op_i evaluated on the ℓ inputs yields into out_i . If all these checks pass, it outputs 1; otherwise, it outputs 0.

5.4.2 The Judge Smart Contract \mathcal{C}

Now that we have seen the algorithms $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$, and $\text{Judge}^{\mathcal{H}}$, we are ready to construct the judge smart contract \mathcal{C} who will be modeled as a hybrid functionality. It interacts with the parties \mathcal{S} and \mathcal{R} , the global ledger \mathcal{L} , and random oracle \mathcal{H} .

Hybrid Functionality \mathcal{C} for the judge contract
<p>The ideal functionality \mathcal{C} acts as a judge smart contract for session id id and interacts with the global \mathcal{L} functionality and the parties \mathcal{S} and \mathcal{R}. It locally stores addresses $pk_{\mathcal{S}}$ and $pk_{\mathcal{R}}$, price p, commitment c, decryption key k, Merkle tree root hashes r_z, r_ϕ and state s.</p>
Initialize
<p>(Step 1) Upon receiving $(\text{init}, id, p, c, r_\phi, r_z)$ from \mathcal{S} with $p \in \mathbb{N}$, store r_ϕ, r_z, p, c, output $(\text{initialized}, id, p, r_\phi, r_z, c)$, set $s = \text{initialized}$ and proceed to the reveal phase.</p> <p>(Step 2) Upon receiving (accept, id) from \mathcal{R} when $s = \text{initialized}$, send $(\text{freeze}, id, \mathcal{R}, p)$ to \mathcal{L}. If it responds with $(\text{frozen}, id, \mathcal{R}, p)$, set $s = \text{active}$ and output $(\text{accepted}, id)$.</p>
Reveal
<p>(Step 3) Upon receiving $(\text{reveal}, id, d, k)$ from sender \mathcal{S} when $s = \text{active}$ and $\text{Open}(c, d, k) = 1$, send $(\text{revealed}, id, d, k)$ to all parties and set $s = \text{revealed}$. Then proceed to the payout phase.</p> <p>Otherwise if no such message from \mathcal{S} was received, send message $(\text{unfreeze}, id, p, \mathcal{R})$ to \mathcal{L} and abort.</p>
Payout
<p>(Step 4) Upon receiving a message m from the receiver \mathcal{R} when $s = \text{revealed}$ set $s = \text{finalized}$ and do the following:</p> <ul style="list-style-type: none"> • If $m = (\text{complain}, id, \pi)$ s.t. $\text{Judge}^{\mathcal{H}}(k, r_z, r_\phi, \pi) = 1$ send $(\text{unfreeze}, id, p, \mathcal{R})$ to \mathcal{L}, $(\text{not sold}, id)$ to \mathcal{S} and terminate. • Otherwise, send $(\text{unfreeze}, id, p, \mathcal{S})$ to \mathcal{L}, (sold, id) to \mathcal{S} and terminate. <p>(Step 5) Upon receiving message $(\text{finalize}, id)$ from the sender \mathcal{S}, when $s = \text{revealed}$, send message $(\text{unfreeze}, id, p, \mathcal{S})$ to \mathcal{L}. Then output (sold, id) to \mathcal{S} and terminate.</p>

5.4.3 The Witness Selling Protocol Π_{FairSwap}

Now we can formally construct our protocol Π_{FairSwap} by using the three algorithms $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$, and $\text{Judge}^{\mathcal{H}}$. The protocol consists of the judge contract and the specification of the behavior of the two honest parties, \mathcal{S} and \mathcal{R} . In order to formally define the functions provided by the judge smart contract \mathcal{C} , we model it as an ideal functionality \mathcal{C} . The full description of \mathcal{C} and the specification of the protocol is given below. Our protocol proceeds in three phases, thereby closely following the structure of the smart contract \mathcal{C} . In the first step in the initialization phase, \mathcal{C} takes as input from \mathcal{S} , the root elements r_z , and r_ϕ as well as the commitment c . \mathcal{R} receives z directly from \mathcal{S} , and r_z, r_ϕ from \mathcal{C} . If these roots are computed correctly, then \mathcal{R} accepts the contract. Additionally, if both parties agree and \mathcal{R} has sufficient funds, p coins are locked for this execution of the fair exchange protocol. Only after this phase is successfully executed, the judge contract is considered active. If, during this phase, some party decides to abort the execution, this is not considered malicious.

Protocol Π_{FairSwap}
The protocol consists of descriptions of the behavior of the honest sender \mathcal{S} and receiver \mathcal{R} .
Initialize
<p>\mathcal{S}: Upon receiving input $(\text{sell}, id, \phi, p, x)$ from the environment \mathcal{Z} in step 1, \mathcal{S} samples $k \leftarrow \text{Gen}(1^\kappa)$, computes $(c, d) \leftarrow \text{Commit}(k)$ and $z = \text{Encode}^{\mathcal{H}}(\phi, x, k)$. Then he sends $(\text{sell}, id, z, \phi, c)$ to \mathcal{R} and $(\text{init}, id, p, c, r_\phi, r_z)$ to \mathcal{C}, where $r_\phi = \text{root}(\text{Mtree}^{\mathcal{H}}(\phi))$ and $r_z = \text{root}(\text{Mtree}^{\mathcal{H}}(z))$. Then he continues to the reveal phase.</p> <p>\mathcal{R}: Upon receiving input (buy, id, ϕ) from the environment \mathcal{Z} in step 2, \mathcal{R} checks if he received message (sell, id, z, c) from \mathcal{S} in step 1 and computes $r_z = \text{root}(\text{Mtree}^{\mathcal{H}}(z))$ and $r_\phi = \text{root}(\text{Mtree}^{\mathcal{H}}(\phi))$. Upon receiving $(\text{init}, id, p, c, r_\phi, r_z)$ from \mathcal{C}, \mathcal{R} responds with (accept, id) and proceeds to the reveal phase.</p>
Reveal
<p>\mathcal{S}: Upon receiving (active, id) from \mathcal{C}, \mathcal{S} responds with $(\text{reveal}, id, d, k)$ and proceeds to the payout phase. If no (active, id) message was received from \mathcal{C} in the third step, he instead terminates the protocol.</p>

<p>\mathcal{R} : Upon receiving $(\text{revealed}, id, d, k)$ from \mathcal{C}, \mathcal{R} proceeds to the payout phase. Otherwise, if no $(\text{revealed}, id, d, k)$ message was received from \mathcal{C} in step 4, \mathcal{R} terminates the protocol.</p>
Payout
<p>\mathcal{R}: The receiver runs $(x, \pi) = \text{Extract}^{\mathcal{H}}(\phi, z, k)$. If $\pi = \perp$, he sends message $(\text{finalize}, id)$ to \mathcal{C}, outputs (bought, id, x) to the environment \mathcal{Z} and terminates the protocol execution. Otherwise (if $\pi \neq \perp$) he sends $(\text{complain}, id, \pi)$ instead.</p> <p>\mathcal{S}: Upon receiving (sold, id) or $(\text{not sold}, id)$ from \mathcal{C}, \mathcal{S} outputs this message and terminates the protocol. If no message has been received in step 4, he sends $(\text{finalize}, id)$ to \mathcal{C}.</p>

In the reveal key phase, the contract expects \mathcal{S} to reveal the key k , which allows verifying the commitment c . If \mathcal{S} fails to send the **reveal** message, he is considered malicious, and \mathcal{R} can get his money back. Otherwise, if \mathcal{S} revealed the key, \mathcal{R} can decode the witness x by running $(x, \pi) = \text{Extract}^{\mathcal{H}}(z, \phi, k)$. In the next phase, the payout of the coins can be triggered. If the witness is valid (i.e., $\pi = \perp$) \mathcal{R} sends message $(\text{finalize}, id)$ to \mathcal{C} , which will trigger the smart contract to unfreeze the coins in \mathcal{S} 's favor. If instead $\text{Extract}^{\mathcal{H}}$ output a valid complaint, \mathcal{R} sends a message $(\text{complain}, id, \pi)$ to \mathcal{C} . If $\phi(x) \neq 1$, the verification algorithm $\text{Judge}^{\mathcal{H}}(k, r_z, r_\phi, \pi)$ will output 1 and thus accept the complaint and all coins are payed back to \mathcal{R} . If \mathcal{R} sends neither message, \mathcal{S} can call the judge contract in step 5, to trigger the payout of coins.

5.5 FairSwap Security Proof

Recall that in the UC framework, the security of a cryptographic protocol Π_{FAIRSWAP} is analyzed by comparing its real world execution with an idealized protocol running in an ideal world. Π_{FAIRSWAP} is attacked by an adversary \mathcal{A} , who can corrupt some of these parties. In the *ideal world*, parties interact with $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$, which has an interface to the ideal world adversary Sim . In both the real and ideal world, the environment \mathcal{Z} provides the inputs for all parties and receives their outputs. A protocol Π_{FAIRSWAP} is said to be *UC-secure* if the environment \mathcal{Z} cannot distinguish whether it is interacting with the ideal or real world. This implies that Π_{FAIRSWAP} is at least as secure as the ideal functionality. To simplify our presentation, we omit

5 Moving Complex Computation Off-Chain

session identifiers and the sub-session identifiers (typically denoted with sid and $ssid$) and use instead of the contract identifier id to uniquely distinguish sessions. In practice, the contract identifier may correspond to the contract address.

Formal Security Definition We consider a protocol Π_{FAIRSWAP} with access to the judge contract functionality \mathcal{G}_{jc} , the global random oracle \mathcal{H} and the global ledger functionality \mathcal{L} . Following the notation introduced in Section 2.4, we denote the real world execution where the environment \mathcal{Z} interacts with a protocol Π_{FAIRSWAP} and an adversary \mathcal{A} on input 1^κ and auxiliary input $x \in \{0, 1\}^*$ as

$$\text{HYBRID}_{\Pi_{\text{FAIRSWAP}}, \mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H}}^{\mathcal{Z}, \mathcal{A}}(\kappa, x).$$

In the ideal world, the parties do not interact with each other but only forward their inputs to an ideal functionality $\mathcal{F}_{\text{cfe}}^\mathcal{L}$. In this setting we will call the adversary a *simulator* Sim and denote the above output as

$$\text{IDEAL}_{\mathcal{F}_{\text{cfe}}^\mathcal{L}, \mathcal{L}, \mathcal{H}}^{\mathcal{Z}, \text{Sim}}(\kappa, x).$$

Given these two random variables, we can now define the security of our protocol Π_{FAIRSWAP} as follows.

Definition 7 (GUC security of Π_{FAIRSWAP}). *Let $\kappa \in \mathbb{N}$ be a security parameter, Π_{FAIRSWAP} be a protocol in the $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$ -hybrid world. Π_{FAIRSWAP} is said to GUC realize $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ in the $(\mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H})$ -hybrid world if for every PPT adversary \mathcal{A} attacking Π_{FAIRSWAP} there exists a PPT algorithm Sim , such that the following holds for all PPT environments \mathcal{Z} and for all $x \in \{0, 1\}^*$:*

$$\text{IDEAL}_{\mathcal{F}_{\text{cfe}}^\mathcal{L}, \mathcal{L}, \mathcal{H}}^{\mathcal{Z}, \text{Sim}}(\kappa, x) \approx_c \text{HYBRID}_{\Pi_{\text{FAIRSWAP}}, \mathcal{G}_{\text{jc}}, \mathcal{L}, \mathcal{H}}^{\mathcal{Z}, \mathcal{A}}(\kappa, x)$$

We are now ready to formally state the security of our protocol Π_{FAIRSWAP} in this GUC-style security notion.

Theorem 2. *The two-party protocol Π_{FAIRSWAP} securely emulates the ideal fair exchange functionality $\mathcal{F}_{\text{cfe}}^\mathcal{L}$ in the judge smart contract $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world, where the global functionality \mathcal{H} is modeled as a restricted programmable and observable random oracle and the global functionality \mathcal{L} models the ledger.*

5.5.1 Informal Security Analysis

The protocol terminates either after four steps, in the payout phase, after \mathcal{R} sends the `finalize` or `complain` message or in the fifth step after \mathcal{S} sent `finalize`. We

5 Moving Complex Computation Off-Chain

distinguish the following termination cases for the protocol with an active judge contract and at least one honest party:

No abort: This case occurs when both parties act honestly. In this case, the protocol terminates in the **payout** phase, after \mathcal{R} sends the **finalize** message to \mathcal{C} .

\mathcal{S} aborts: In case \mathcal{S} does not reveal the key k , \mathcal{C} will terminate in the reveal phase and make sure that \mathcal{L} assigns all coins to \mathcal{R} .

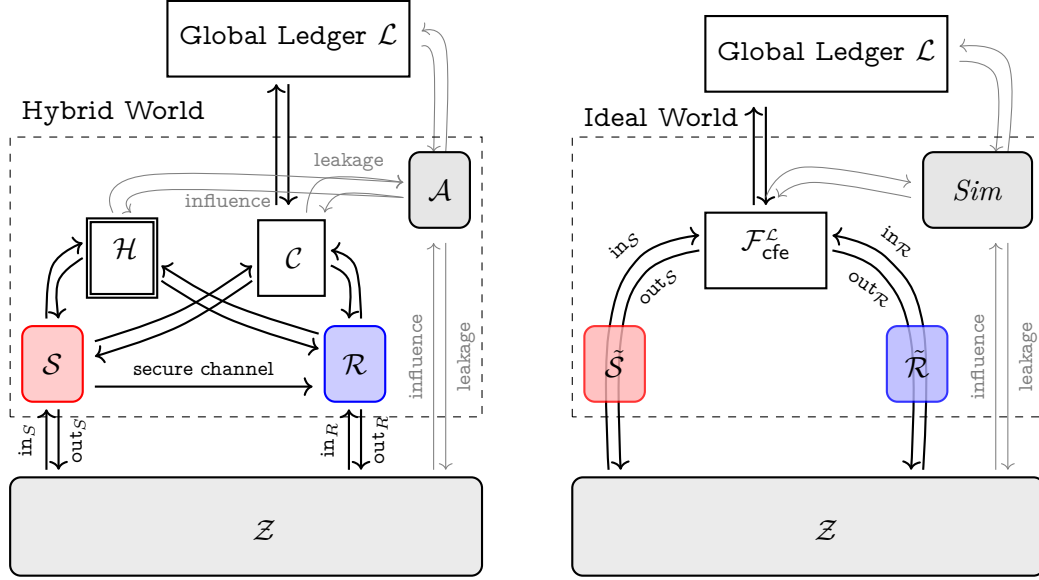
\mathcal{R} aborts: This case occurs when \mathcal{R} does not react anymore after the key was revealed. In the fifth step, \mathcal{S} will then send $(\text{finalize}, id)$ to \mathcal{C} , and the coins will be sent to \mathcal{S} .

Sender Fairness. Sender fairness means that \mathcal{R} must not learn the witness x unless the honest sender \mathcal{S} is guaranteed to be paid. From the secrecy property of our encoding scheme and the hiding property of the commitment, it follows directly that \mathcal{R} cannot read the content of the encoded witness before \mathcal{S} publishes the encoding key k . At the point, when k is revealed, the coins have been successfully frozen for the execution of the smart contract \mathcal{C} . Now, that the exchange of the witness is initiated, an honest \mathcal{S} is guaranteed to receive the payment, even if \mathcal{R} aborts. Lastly, it remains to show that a malicious \mathcal{R} cannot forge a proof π , which is accepted by the judge contract, although \mathcal{S} behaved honestly and $\phi(x) = 1$. Forging such a proof would require \mathcal{R} to forge a Merkle proof over a false element of z . Informally speaking, this is not possible unless he finds a collision in the hash function \mathcal{H} .

Receiver Fairness. If \mathcal{S} sends the encoding z , \mathcal{R} continues with the protocol until the coins are frozen for the execution of the smart contract \mathcal{C} . To prove fairness for an honest receiver \mathcal{R} , we have to show that a malicious sender \mathcal{S} cannot send a wrong witness $x' \notin L$ such that \mathcal{R} is not able to generate a correct proof of misbehavior, which is accepted by the \mathcal{C} contract. In order to successfully execute such an attack, \mathcal{S} must be able to find an encoding z such that $\text{Extract}^{\mathcal{H}}(z, k, \phi) = (x', \pi')$ but the judge on input of π does not accept the complaint. The probability of \mathcal{S} finding such values is negligible since this would require him to break collision resistance of the underlying hash function. Therefore, \mathcal{R} is guaranteed, that as soon as \mathcal{S} publishes k , he will either receive the witness x with $\phi(x) = 1$, or he has the guarantee that by executing \mathcal{C} on a valid proof of misbehavior he will get all coins. Therefore Π_{FAIRSWAP} satisfies receiver fairness.

5.5.2 Formal GUC Security Proof

In order to prove Theorem 2, we show that Π_{FAIRSWAP} is a secure realization of the ideal smart sale functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. We need to show that the ideal world



(a) $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world execution of Π_{FAIRSWAP} with S, R and A (b) Execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with dummy parties \tilde{S} and \tilde{R} and Sim

Figure 5.6: Setup of a Simulation with honest parties

(the execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with dummy parties \tilde{S} and \tilde{R} and the ideal adversary Sim) is indistinguishable from the hybrid world. In our case, the hybrid world is the execution of Π_{FAIRSWAP} with parties S, R , and an adversary A where each party interacts with the hybrid functionalities \mathcal{C}^S and \mathcal{H} . Figure 5.6 depicts the setup of the security proof. The PPT environment \mathcal{Z} distinguishes whether it interacts with the hybrid world execution of the protocol Π_{FAIRSWAP} (cf. Figure 5.6 (a)) or with the ideal execution of the functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ (cf. Figure 5.6 (b)). In the ideal world, the parties \tilde{R} and \tilde{S} are so-called dummy parties that only forward the in- and outputs of \mathcal{Z} to the ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ whereas in the hybrid world the parties run the code of protocol Π_{FAIRSWAP} . \mathcal{Z} can use the leakage information from the adversary A (respectively the ideal adversary Sim) or actively influence

⁸As our hybrid functionality \mathcal{C} models a contract, we indicate this relation by using the earlier introduced double-line depiction.

5 Moving Complex Computation Off-Chain

the execution to distinguish the two worlds. Additionally, it selects the inputs for the two parties and learns its outputs. Lastly, \mathcal{Z} can corrupt any of the parties (using the adversary) to learn any internal values, and all messages sent to and from the party. We will consider these cases in detail later. To prevent \mathcal{Z} from distinguishing the executions, we need to construct a simulator, that outputs all messages such that it looks like the hybrid world execution to the Environment \mathcal{Z} . Specifically, Sim needs to ensure that the outputs of the parties are identical in the hybrid world and the simulation. Even with corrupted parties, he needs to generate an indistinguishable transcript of the real world execution while ensuring that the global functionality \mathcal{L} blocks or unblocks money in the same rounds. In order to formally prove Theorem 2, we need to consider four cases, the protocol execution with two honest parties, execution with a malicious sender \mathcal{S}^* , execution with a dishonest receiver \mathcal{R}^* and the case where both parties are corrupt. All of the described termination cases (cf. Section 5.4) provide seller and buyer fairness as defined in Section 5.3.

Simplifications and Notation. Whenever the simulator Sim simulates the execution of \mathcal{C} on some input of message m we write $m' \leftarrow \mathcal{C}(m)$ to indicate that the message m' is the output of \mathcal{C} after m was received. Simulation of this execution includes leaking these messages m, m' to the environment, and sending m' to the corrupted parties according to the behavior of \mathcal{C} . Note, that \mathcal{C} is only internally simulated by Sim and does not freeze/unfreeze coins in \mathcal{L} . Whenever parties send messages to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and \mathcal{C} , the environment (over the adversary) has the power to delay these messages by time Δ . We will not argue about this power in detail during the simulation since we make the following simplifying assumption. In every step (1) - (5), the adversary may instruct the ideal functionality (over the *influence port*) by how much time the message is delayed. Using this knowledge, Sim will ensure that $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ is always delayed by the same amount of time as the \mathcal{C} functionality would be. This ensures that \mathcal{Z} cannot distinguish the real world for the simulation, using this influence. This simplification allows us to construct the simulators without mentioning this influence explicitly in every step. To simplify complex steps in our proofs, we sometimes use a sequence of simulation games. This technique is often used in simulation based proofs to show indistinguishability. Instead of showing indistinguishability of the real world execution Π and ideal world simulation with Sim immediately, we construct the experiments $Game_0, \dots, Game_n$. We call the real world execution $Game_0$, and $Game_n$ is the final UC simulation. The intermediate games are hybrid simulations, where each

$Game$ is one step closer to the ideal world simulation, but the simulator in these intermediate games additionally controls the in- and outputs of the honest parties. By showing that for each $i \in [n - 1]$ that $Game_i$ is computationally indistinguishable from $Game_{i+1}$ we show that the real world execution is indistinguishable from the ideal world simulation, i.e., $Game_0 \approx_c Game_n$.

Simulation without corruptions

Simulation of the protocol execution with an honest seller and honest receiver is a special case, in which the dummy parties $\tilde{\mathcal{S}}$ and $\tilde{\mathcal{R}}$ will forward all messages from \mathcal{Z} to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the ideal world (as depicted in Figure 5.6 (b)). The simulation, in this case, is straight forward since Sim^{honest} will be only required to generate a transcript of all messages of the execution of Π_{FAIRSWAP} towards the adversary and thus \mathcal{Z} . This includes the simulation of the first protocol message, send from \mathcal{S} to \mathcal{R} , and all following interactions with \mathcal{C} and \mathcal{H} . Note that the communication between the honest \mathcal{S} and \mathcal{R} is private, and \mathcal{Z} cannot read the content of this message, but only see if a message was sent.

Claim 1. *There exists a efficient algorithm Sim^{honest} such that for all PPT environments \mathcal{Z} , that **do not corrupt any party** it holds that the execution of Π_{FAIRSWAP} in the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world in the presence of adversary \mathcal{A} is computationally indistinguishable from the ideal world execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with the ideal adversary Sim^{honest} .*

Proof. We define a simulator Sim^{honest} , which internally runs \mathcal{C} and has oracle access to \mathcal{H} .

Simulator Sim^{honest} without corruptions

1. If \mathcal{S} starts the execution with $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the first step Sim^{honest} learns id, p, ϕ from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. Then Sim^{honest} , selects $k \leftarrow \{0, 1\}^\mu$, sets $x^* = 1^{n \times \lambda}$ and computes $\text{Encode}^{\mathcal{H}}(x^*, k, \phi) = z$. He computes $r_\phi = \text{root}(\text{Mtree}^{\mathcal{H}}(\phi))$, $r_z = \text{root}(\text{Mtree}^{\mathcal{H}}(z))$ and $(c, d) \leftarrow \text{Commit}(k)$. Now he simulates the execution of Π_{FAIRSWAP} by running $(\text{initialized}, id, p, c, r_\phi, r_z) \leftarrow \mathcal{C}(\text{init}, id, p, c, r_\phi, r_z)$.
2. If \mathcal{R} sends $(\text{buy}, id, \mathcal{R})$ in the execution with $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the second step, Sim^{honest} runs $(\text{accepted}, id) \leftarrow \mathcal{C}(\text{accept}, id)$ to simulate the acceptance by \mathcal{R} . If instead \mathcal{R} does not send this message in the ideal world, Sim^{honest} simulates the automatic refund of p coins in \mathcal{C} and terminates the simulation.

3. In the reveal phase, Sim^{honest} runs $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$.
4. In the payout phase, Sim^{honest} waits for $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ to unfreeze the coins. If $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ unfroze them in favor of \mathcal{S} , Sim^{honest} simulates the execution of $(\text{sold}, id) \leftarrow \mathcal{C}(\text{finalize}, id)$. On the other hand, if the coins are unfrozen in the name of \mathcal{R} , Sim^{honest} simulates a complain by \mathcal{R} . Specifically, he generates a complain about the output of ϕ i.e., that the output of g_m does not equal 1^a . Then terminate the simulation.

^aSpecifically, $\pi = ((\phi_m, \text{Mproof}^{\mathcal{H}}(m, \phi_m, \text{Mtree}^{\mathcal{H}}(\phi))), (0, \text{Mproof}^{\mathcal{H}}(m, 0, \text{Mtree}^{\mathcal{H}}(z))))$

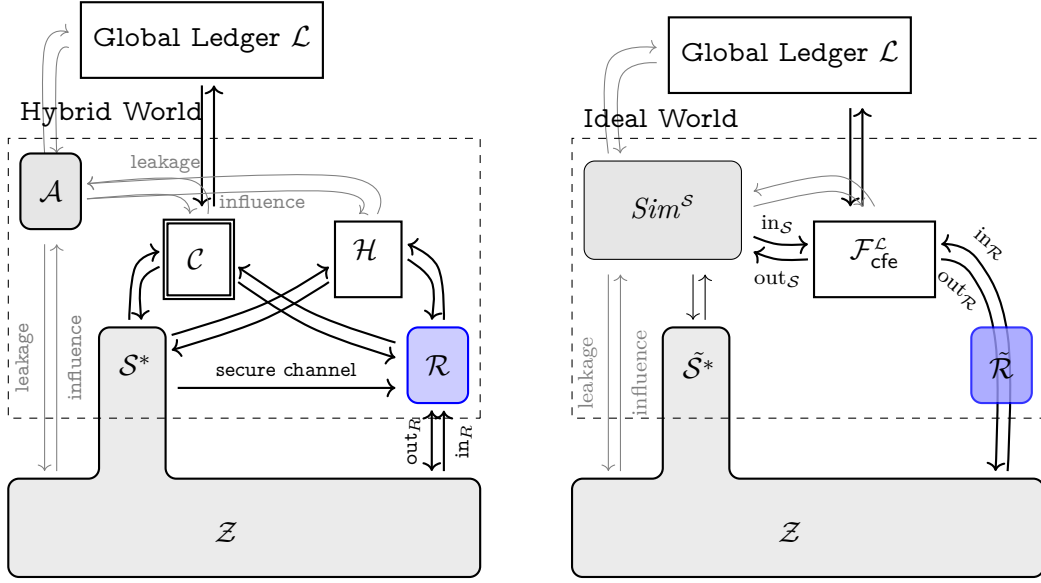
Running Sim^{honest} in the $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ ideal world is indistinguishable from the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world execution to \mathcal{Z} unless \mathcal{Z} learns z and extracts $x' \neq x$ (using k , which he learns in the **reveal** phase). But we can show that this only happens if z breaks the hiding property of the randomized Merkle tree commitment, which is computationally hard, as explained in Section 5.2.4. This way \mathcal{Z} does not learn any decoding of z , except for the last element which equals 0 if Sim^{honest} simulates a complaint by \mathcal{R} . Note, that in the honest case, both parties will follow the protocol, thus most complain cases cannot happen. The only possible complaint case occurs when the environment inputs a false file and hash to the parties. \square

Simulation with a malicious sender

Simulation with a corrupted sender is slightly more tricky than the simulation with two honest parties. The simulator, in this case, needs to simulate the transcript of Π_{FAIRSWAP} and, additionally, all outputs of the corrupted sender towards $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and \mathcal{Z} . This means whenever \mathcal{Z} sends a message through the corrupted dummy party \mathcal{S}^* , it is sent to $Sim^{\mathcal{S}}$ directly. Using these inputs, $Sim^{\mathcal{S}}$ internally simulates the execution of Π_{FAIRSWAP} while interacting with $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the name of \mathcal{S}^* . Figure 5.7 shows the setup of this simulation.

Claim 2. *There exists a efficient algorithm $Sim^{\mathcal{S}}$ such that for all PPT environments \mathcal{Z} , that **only corrupt the sender** it holds that the execution of Π_{FAIRSWAP} in the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world in the presence of adversary \mathcal{A} is computationally indistinguishable from the ideal world execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with the ideal adversary $Sim^{\mathcal{S}}$.*

Proof. Since the simulation in the presence of a corrupted sender is not straight forward, we construct a sequence of two simulation games $Game_1$ and $Game_2$,


 (a) $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world execution of Π_{FAIRSWAP} with \mathcal{S}^* , \mathcal{R} and \mathcal{A}

 (b) Execution of \mathcal{F}_{cfe}^L with dummy parties $\tilde{\mathcal{S}}^*$ and $\tilde{\mathcal{R}}$ and Sim^S

 Figure 5.7: Simulation with corrupted sender \mathcal{S}^* and honest receiver \mathcal{R}

where the simulator of $Game_2$ equals the simulator Sim^S from the ideal world execution. But before we construct this ideal adversary, we will start with a slightly modified Experiment $Game_1$, in which the simulator Sim_1^S holds the private inputs of the honest receiver and generates messages on his behalf. We will first sketch this simulator Sim_1^S and argue that the execution of Π_{FAIRSWAP} in the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world is indistinguishable to the execution of $Game_1$ and then show how to change the simulator, such that it runs in the ideal world ($Game_2$) and again show indistinguishability.

Simulator Sim_1^S

1. Upon receiving message $(\text{init}, id, p, c, r_z, r_\phi)$ in the first step, Sim_1^S simulates the execution of the hybrid functionality by sending $(\text{initialized}, p, c, r_\phi, r_z) = \mathcal{C}(\text{init}, id, p, c, r_z, r_\phi)$ to \mathcal{S}^* . If \mathcal{S}^* did not send the message (init) , Sim_1^S aborts the simulation. Otherwise, if message $(\text{sell}, id, z, \phi, c)$ was also received from \mathcal{S}^* in the first step, Sim_1^S sets $x^* = 1^{n \times \lambda}$ and sends $(\text{init}, id, \phi, p, x^*)$ to \mathcal{F}_{cfe}^L .

5 Moving Complex Computation Off-Chain

2. In the second step Sim_1^S waits to receives $(\text{buy}, id, \mathcal{R})$ from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$, which means that the p coins were frozen in \mathcal{L} . To simulate that this was performed by the judge smart contract, Sim_1^S runs $(\text{active}, id) = \mathcal{C}(\text{accept}, id)$. If no message from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ was received in step 2, Sim_1^S terminates the simulation.
3. Upon receiving $(\text{reveal}, id, d, k)$ from \mathcal{S}^* in step 3, such that $\text{Open}(c, d, k) = 1$, Sim_1^S triggers $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ to output x^* to \mathcal{R} . Additionally, Sim_1^S locally simulates the execution of Π_{FAIRSWAP} by running $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$. Since Sim_1^S controls the in- and outputs of \mathcal{R} , he will exchange the output message to the environment from (bought, id, x^*) to (bought, id, x) , where x is the extracted witness $(x, \pi) \leftarrow \text{Extract}^{\mathcal{H}}(\phi, z, k)$. If no message $(\text{reveal}, id, d, k)$ from \mathcal{S}^* is received, send (abort, id) to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the name of \mathcal{S}^* and simulate the refund in the hybrid world by running \mathcal{C} until it terminates.
4. If $\pi = \perp$, Sim_1^S simulates the acceptance by an honest receiver by running $(\text{sold}, id) \leftarrow \mathcal{C}(\text{finalize}, id)$ and send (abort, id) to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. He immediately triggers the unfreezing of \mathcal{S} coins in the ledger \mathcal{L} and outputs (sold, id) to \mathcal{S}^* . If instead $\pi = 1$, Sim_1^S needs to simulate a complaint. He does this by running $(\text{not sold}, id) \leftarrow \mathcal{C}(\text{complain}, id, \pi)$ and letting $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ terminate normally. Then Sim_1^S outputs $(\text{not sold}, id)$ to \mathcal{S}^* and terminates.

The output (bought, id, x) of \mathcal{R} in step (3) is identical to the real world execution, since it is computed in the same way, as the extraction of z using key k . The same holds for the outputs to \mathcal{S}^* in step (1), (2) and (4) since they are generated by the simulation of \mathcal{C} , with the honestly generated inputs of \mathcal{R} . The environment \mathcal{Z} will only be able to distinguish the real world execution from the execution of $Game_1$ if messages are sent in different steps or coins are frozen/ unfrozen differently in \mathcal{L} . Sim_1^S makes sure to simulate messages of \mathcal{C} , and $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the same step as \mathcal{Z} instructs him on the influence port. Only in the case where an honest receiver would send the `finalize` message, Sim^S makes sure that the `abort` is received by $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the same step as it unfreezes the coins for \mathcal{S} . This is necessary since the input to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ is not correct, and upon checking, $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ would not assign the coins to \mathcal{S} . But by immediately aborting and triggering the unfreeze, Sim_1^S can successfully simulate the case of no complaint. Since $Game_1$ does not capture the full functionality of Sim_1^S , we need to adapt Sim_1^S further to work in the $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ world. Mainly, Sim^S must not be able to control inputs and outputs of honest parties like depicted in Figure 5.7. Instead, it needs to utilize the observability

5 Moving Complex Computation Off-Chain

functions of the random oracle and learn the witness x from the messages send by \mathcal{S}^* in the first step of the protocol and input this witness to the ideal functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. Instead of constructing a new simulator, we will only highlight the differences to the previous simulator $Sim_1^{\mathcal{S}}$.

Simulator $Sim^{\mathcal{S}}$ for corrupted sender

1. Upon receiving message $(\text{sell}, id, z, \phi, c)$ over \mathcal{S}^* in the first step, $Sim^{\mathcal{S}}$ learns Q_{id} from querying $\mathcal{H}(\text{observe})$ and checks if $(k||d, c) \in Q_{id}$. If such a query exists in Q_{id} $Sim^{\mathcal{S}}$ runs $(x, \pi) = \text{Extract}^{\mathcal{H}}(\phi, z, k)$. If no such query exists, or \mathcal{S}^* never send sell , set $x = 1^{n \times \lambda}$.

Upon receiving $(\text{init}, id, p, q, c, r_z, r_\phi)$ through \mathcal{S}^* in the first step, send $(\text{init}, id, \phi, p, q, x)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. Simulate the execution of the hybrid functionality by sending $(\text{initialized}, p, c, r_\phi, r_z) = \mathcal{C}(\text{init}, id, p, c, r_z, r_\phi)$ to \mathcal{S}^* .

If \mathcal{S}^* did not send both messages sell and init , $Sim^{\mathcal{S}}$ aborts the simulation.

2. When $Sim^{\mathcal{S}}$ receives $(\text{buy}, id, \mathcal{R})$ from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$, it means that the p coins were frozen in \mathcal{L} . To simulate that this was performed by the judge smart contract, $Sim_1^{\mathcal{S}}$ runs $(\text{active}, id) = \mathcal{C}(\text{accept}, id)$. If no message from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ was received until step 3, $Sim_1^{\mathcal{S}}$ terminates the simulation.
3. Upon receiving $(\text{reveal}, id, d, k)$ from \mathcal{S}^* in step 3, such that $\text{Open}(c, d, k) = 1$, $Sim_1^{\mathcal{S}}$ triggers $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ to output x to \mathcal{R} . Additionally $Sim_1^{\mathcal{S}}$ runs $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$ to simulate the execution of Π_{FAIRSWAP} .

If no message $(\text{reveal}, id, d, k)$ from \mathcal{S}^* is received, send (abort, id) to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the name of \mathcal{S}^* and simulate the refund in the hybrid world by running \mathcal{C} until it terminates.

4. Upon receiving (sold, id) from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$, $Sim^{\mathcal{S}}$ simulates the execution of \mathcal{C} by outputting $(\text{sold}, id) \leftarrow \mathcal{C}(\text{finalize}, id)$ to \mathcal{S}^* . Then $Sim^{\mathcal{S}}$ terminates.

If instead he receives $(\text{not sold}, id)$ from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$, $Sim^{\mathcal{S}}$ runs $(\text{not sold}, id) \leftarrow \mathcal{C}(\text{complain}, id, \pi)$ and terminates.

Since $Sim^{\mathcal{S}}$ cannot control the output of \mathcal{R} anymore, it needs to guarantee that $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ outputs (bought, id, x) in step (3) to the honest receiver \mathcal{R} . Therefore, $Sim^{\mathcal{S}}$

5 Moving Complex Computation Off-Chain

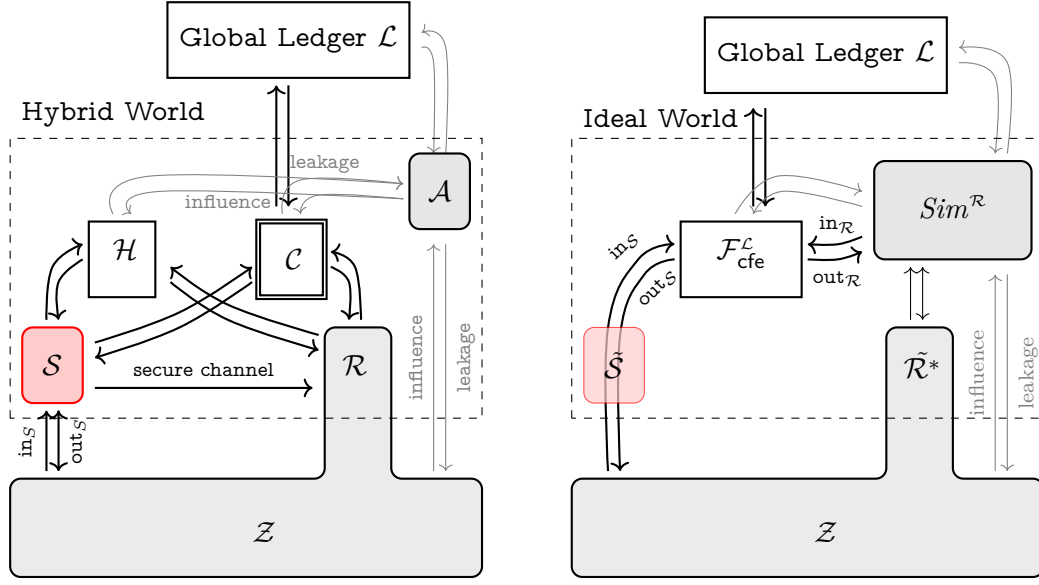
has to input x to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the first step of the execution. But since $\text{Sim}^{\mathcal{S}}$ does not know x he has to learn it from the inputs of \mathcal{S}^* . Specifically, he uses the observability property of the global random oracle \mathcal{H} to get a list Q_{id} of all queries to \mathcal{H} that were made by the environment (directly or over some adversary). We distinguish the following cases now: (a) Either the commitment is not correct, in this case $\text{Sim}^{\mathcal{S}}$ cannot learn k from Q_{id} (this is identical to the case that k was programmed, which makes the commitment invalid), or (b) the commitment was done correctly. In case (a) the execution of the real world protocol will fail just as the simulation with overwhelming probability, since \mathcal{Z} will not be able to provide an opening to commitment c , such that the opening is accepted by \mathcal{C} , except if \mathcal{Z} guesses c, d such that later \mathcal{H} upon being queried $k||d$ responds with exactly with c . Since \mathcal{H} selects the query response randomly from $\{0, 1\}^{\mu}$ this only happens with probability $\frac{1}{2^{\mu}}$, which is negligible for large μ . Case (b) occurs when the tuple $(k||d, c)$ is stored in the set Q_{id} . This allows $\text{Sim}^{\mathcal{S}}$ to run the extraction algorithm, just like the honest sender will and recover x . If the commitment is opened correctly in step (3), the honest receiver in the real world would output x , just as in the interaction with $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in our case. It is only possible to distinguish these two cases if \mathcal{Z} managed to find a collision, i.e., \mathcal{Z} must commit to one key and open to another key, i.e., find a $(k, d), (k', d')$ such that $\text{Open}(c, d, k) = \text{Open}(c, d', k') = 1$. But from the binding property of the commitment scheme it follows that this is not possible except with negligible probability. Thus, we have shown that the hybrid and the ideal world are indistinguishable to the environment \mathcal{Z} if the commitment of the key is binding. This concludes the proof for the case of a malicious sender. \square

Simulation with a malicious receiver

Next, we will show security against malicious receivers (denoted as \mathcal{R}^*). The setup of the simulation is symmetrical to the one with a malicious sender and is depicted in Figure 5.8.

Claim 3. *There exists an efficient algorithm $\text{Sim}^{\mathcal{R}}$ such that for all PPT environments \mathcal{Z} , that **only corrupt the receiver** it holds that the execution of Π_{FAIRSWAP} in the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world in the presence of adversary \mathcal{A} is computationally indistinguishable from the ideal world execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with the ideal adversary $\text{Sim}^{\mathcal{R}}$.*

Proof. The main challenge for $\text{Sim}^{\mathcal{R}}$ in this proof is to provide the encoding z without knowledge of the witness x in the first step and to present key k in the



(a) $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world execution of Π_{FAIRSWAP} with $\mathcal{S}, \mathcal{R}^*$ and \mathcal{A}

(b) Execution of $\mathcal{F}_{cfe}^{\mathcal{L}}$ with dummy parties $\tilde{\mathcal{S}}$ and $\tilde{\mathcal{R}}^*$ and $Sim^{\mathcal{R}}$

Figure 5.8: Simulation against \mathcal{Z} with honest sender \mathcal{S} and malicious receiver \mathcal{R}^*

third step such that the decryption of z yields x . Additionally, the key he provides during the reveal phase has to correctly open a commitment c , which $Sim^{\mathcal{R}}$ has to output in the first step. In order to construct this simulator, we will mainly utilize the programmability property of the global random oracle \mathcal{H} . Again, we construct the simulator using multiple experiments, $Game_1$, $Game_2$, and $Game_3$, which represents the ideal world execution with $\mathcal{F}_{cfe}^{\mathcal{L}}$. We start with the first experiment $Game_1$, in which we give the adversary $Sim_1^{\mathcal{R}}$ the extra power to learn all inputs and give all outputs to honest parties. In our case we require that it can read the input of the dummy party $\tilde{\mathcal{S}}$ in the first step, which is the message $(\text{sell}, id, \phi, p, x)$. Specifically $Sim_1^{\mathcal{R}}$ learns x in this step, which he would not know otherwise. Now we sketch the algorithm of our simulator $Sim_1^{\mathcal{R}}$ with knowledge of x

Simulator $Sim_1^{\mathcal{R}}$

1. The simulation starts when $\tilde{\mathcal{S}}$ sends $(\text{sell}, id, \phi, p, x)$ to $\mathcal{F}_{cfe}^{\mathcal{L}}$. $Sim_1^{\mathcal{R}}$ simulates the execution of Π_{FAIRSWAP} by randomly sampling a key k , encoding $\phi(x)$ to z and sending $(\text{sell}, id, z^*, \phi, c^*)$ to \mathcal{R}^* . When the functionality outputs

5 Moving Complex Computation Off-Chain

$(\text{sell}, id, \phi, p, \mathcal{S})$ to the corrupted receiver $Sim^{\mathcal{R}}$ internally runs \mathcal{C} and outputs $(\text{init}, id, p, c^*, r_\phi, r_z)$ instead (where c is the commitment and r_ϕ and r_z are the Merkle root hashes as defined in the protocol).

2. If the corrupt receiver accepts the exchange, $Sim_1^{\mathcal{R}}$ receives the message (accept, id) from \mathcal{R}^* in step 2. In this case $Sim_1^{\mathcal{R}}$ sends $(\text{buy}, id, \phi, p)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and simulates the judge contract by sending $(\text{active}, id) \leftarrow \mathcal{C}(\text{accept}, id)$ to \mathcal{R}^* .
3. In the reveal phase $Sim_1^{\mathcal{R}}$ simulates the honest sender by outputting $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$ to \mathcal{R}^* .
4. In the payout phase the corrupted receiver can either accept the file, complain or abort the protocol execution altogether. $Sim_1^{\mathcal{R}}$ waits for the message of \mathcal{R}^* in the next step. If he receives a message $(\text{complain}, id, \pi)$ where π is a valid complain, $Sim_1^{\mathcal{R}}$ lets $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ continue, this way the resulting messages of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and $\mathcal{C}(\text{complain}, id, \pi)$ will be identical, namely (sold, id) if $\phi(x) = 1$ or $(\text{not sold}, id)$ otherwise. If $Sim^{\mathcal{R}}$ instead receives a message $(\text{finalize}, id)$, he sends $(\text{abort}, \mathcal{R}^*)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and immediately triggers the unfreezing of p coins. This way the output of (sold, id) and the unfreezing of coins will be indistinguishable to \mathcal{Z} , even when the sent witness is false, i.e., $\phi(x) = 1$. The third case occurs when $Sim_1^{\mathcal{R}}$ does not receive any valid message from \mathcal{R}^* in step 4. In this case, he sends $(\text{abort}, \mathcal{R}^*)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and waits for one step before he triggers the unfreezing of coins. This way, the behavior of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ will be indistinguishable from the judge smart contract of input of $(\text{finalize}, id)$, which an honest sender would always send in step 5.

Thus, the execution of the experiment $Game_1$ running with adversary $Sim_1^{\mathcal{R}}$ is indistinguishable from the real world execution of Π_{FAIRSWAP} with the judge smart contract \mathcal{C} . Note that this simulation is only possible since $Sim_1^{\mathcal{R}}$ learns x in the first step. We will now show how to construct a simulator that does not require this additional input of x but uses the programmability of the random oracle to simulate this knowledge. We will do this in two steps. First, we simulate the key commitment without using the real encoding key and, in a third experiment, also simulate the encoding of z without knowledge of x . We only sketch the main differences to the previous simulator and later give a detailed construction of $Sim^{\mathcal{R}}$, which is the ideal adversary in the execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. The Simulator of Experiment $Game_2$ is very similar to $Sim_1^{\mathcal{R}}$ and only differs in the following way:

Simulator $Sim_2^{\mathcal{R}}$

1. In the first step, $Sim_2^{\mathcal{R}}$ does not generate a commitment $(c, d) \leftarrow \text{Commit}(k)$ but randomly samples $c^* \leftarrow \{0, 1\}^\mu$ and outputs c^* instead of c to the corrupted receiver. At this point \mathcal{Z} cannot distinguish if it received c or c^* , as long as c^* was sampled uniformly at random from the same domain that the random oracle \mathcal{H} uses.
3. In the reveal phase, $Sim_2^{\mathcal{R}}$ now needs to open the commitment and present an opening value d , such that $\text{Open}(c^*, k, d^*) = 1$. $Sim_2^{\mathcal{R}}$ randomly chooses $d^* \leftarrow \{0, 1\}^\kappa$ and sends $(\text{program}, id, k||d^*, c^*)$ to \mathcal{H} to program the random oracle to respond to all oracle queries of $k||d^*$ with c^* .

This will succeed if \mathcal{H} was not queried or programmed on $k||d^*$ before. Since \mathcal{Z} is computationally bounded, it can only guess r and program or query \mathcal{H} on polynomial many points. Additionally, if $Sim_2^{\mathcal{R}}$ chooses an opening value such that the programming fails, he can try again with a different value. Now when \mathcal{Z} checks the opening $\text{Open}(c^*, k, d^*)$, it will try to detect this programming behavior in the simulation, by querying $\mathcal{H}(\text{isPrgrmd}, x||r)$ from this session either over the ideal adversary or a corrupted party. Since $Sim_1^{\mathcal{R}}$ is the ideal adversary and controls all corrupted parties, he can interject all queries to \mathcal{H} from this session and simply send back a false response, to lie about the programmed values. Therefore the result of the simulation of the commitment will be indistinguishable to \mathcal{Z} with overwhelming probability. Therefore the environment cannot distinguish the execution of experiment $Game_2$ from the execution of $Game_3$, except with negligible probability. It remains to show that $Sim^{\mathcal{R}}$ can simulate the encoding z^* without knowledge of x in the first step, such that it is indistinguishable towards \mathcal{Z} . This is possible since the construction of our encoding scheme using the programmable random oracle makes it non-committing for only the ideal adversary $Sim^{\mathcal{R}}$. He proceeds as follows: In the first step he simulates the encoding z by sampling it uniformly at random, i.e., $z = (z_1, \dots, z_m) \xleftarrow{\$} \{0, 1\}^{\mu \times m}$. Upon learning the actual witness $x = (x_1, \dots, x_n)$ $Sim^{\mathcal{R}}$ needs to output a key k such that $\text{Dec}(k, z) = (x)$. Knowing x , he samples $k \leftarrow \{0, 1\}^\kappa$ and programs the random oracle \mathcal{H} to open all decryption queries as follows:

$$\forall i \in m : \mathcal{H}(\text{program}(k||i, o_i \oplus z_i))$$

Should \mathcal{Z} request the response of $(\text{isPrgrmd}, k||i)$ for any $i \in n$ the simulator $Sim^{\mathcal{R}}$ lies to \mathcal{Z} and claims that it was not programmed. This simulation is indistinguishable from the real world execution as long as the programming is not detected by \mathcal{Z} , which happens if it queries $\mathcal{H}(k||i)$ for any $i \in [n]$ or programmed any of these values himself before the programming took place. If the adversary does not know k , this happens only with negligible probability since he can only make polynomially many queries or programming requests to \mathcal{H} . Therefore the execution of experiment $Game_2$ is computationally indistinguishable from experiment $Game_3$. By showing that $Sim^{\mathcal{R}}$ can simulate the encoding z without the knowledge of x , we have completed all necessary steps of the simulation and can construct the ideal world simulator $Sim^{\mathcal{R}}$ and concluded the proof. To give the reader a complete overview of the final simulator for the ideal world execution, we formalize it in detail below, combining the steps of the three experiments above. \square

Simulator $Sim^{\mathcal{R}}$

1. Upon receiving $(\text{sell}, id, \phi, p, \mathcal{S})$ from $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the first step $Sim^{\mathcal{R}}$ randomly samples $z^* \leftarrow \{0, 1\}^{m \times \mu}$. Additionally, he simulates the commitment as $c^* \leftarrow \{0, 1\}^{\mu}$ and sends the message $(\text{sell}, id, z^*, \phi, c^*)$ to \mathcal{R}^* . Next he computes $r_{\phi} = \text{root}(\text{Mtree}^{\mathcal{H}}(\phi))$ and $r_z = \text{root}(\text{Mtree}^{\mathcal{H}}(z^*))$, runs \mathcal{C} on input $(\text{init}, id, p, c^*, r_{\phi}, r_z)$ and sends the output to \mathcal{R}^* .
2. Wait to receive (accept, id) from \mathcal{R}^* in the third step.
 - If no such message is received, $Sim^{\mathcal{R}}$ simulates the refund of locked coins in \mathcal{C} and terminates the simulation.
 - If $Sim^{\mathcal{R}}$ receives the **accept** message he sends message $(\text{buy}, id, \phi, p)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and simulates the activation of \mathcal{C} by sending $(\text{active}, id) \leftarrow \mathcal{C}(\text{accept}, id)$ to \mathcal{R}^* .
3. In the reveal phase $Sim^{\mathcal{R}}$ learns x from the message (bought, id, x) , which $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ sends to \mathcal{R}^* . Then he needs to simulate the messages of the honest sender in the protocol.
 - $Sim^{\mathcal{R}}$ selects k uniformly at random from $\{0, 1\}^{\kappa}$ and for all $i \in [n]$ set $o_i := x_i$ and for all $j \in \{0, \dots, m - n\}$ and $\phi_j := (i, \text{op}_i, I_i)$ computes $o_{n+i} := \text{op}_i(\{o_j\}_{j \in I_i})$.
 - Then map the encoding of z^* and k to the correct values by programming the random oracle \mathcal{H} in the following way: for all $i \in m$ send the messages $(\text{program}(k||i, o_i \oplus z_i))$ to \mathcal{H} . Abort if

the programming fails. Otherwise, from now on querying $\mathcal{H}(k||i)$ results in $r_i = z_i^* \oplus o_i$ such that for all $i \in n$ $\text{Enc}(k, z_i^*) = z_i \oplus \mathcal{H}(k||i) = o_i$ will decode z_i^* to o_i .

- To generate the correct opening for the commitment, $Sim^{\mathcal{R}}$ samples $d \leftarrow \{0, 1\}^\kappa$ and programs the random oracle by sending $\mathcal{H}(\text{program}(k||d, c^*))$. Abort if the programming fails.

Whenever \mathcal{Z} queries \mathcal{H} (over \mathcal{R}^* or $Sim^{\mathcal{R}}$) on the programmed values, $Sim^{\mathcal{R}}$ interjects these queries and instead of forwarding them to \mathcal{H} , responds to them himself so he can pretend that the values are not programmed. Finally, $Sim^{\mathcal{R}}$ runs $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$.

4. $Sim^{\mathcal{R}}$ waits to receive a message from \mathcal{R}^* in step 4.
 - If the message is a valid complaint $(\text{complain}, id, \pi)$ $Sim^{\mathcal{R}}$ runs \mathcal{C} on input of this message. If the judge contract accepts the complaint, he lets $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ continue and terminates.
 - If the message is a false complaint or $(\text{finalize}, id)$ $Sim^{\mathcal{R}}$ instead sends (abort, id) to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ and immediately trigger the following execution including the unfreezing of coins. Then $Sim^{\mathcal{R}}$ terminates.
 - If instead \mathcal{R}^* does not send a message, $Sim^{\mathcal{R}}$ also sends (abort, id) but waits for one step before he triggers the further execution, including the payout of coins. Then $Sim^{\mathcal{R}}$ terminates.

It remains to argue why the probability that the simulation fails is negligible. This case only occurs when $Sim^{\mathcal{R}}$ fails to program the \mathcal{H} because a value was programmed before. To trigger this case, the Environment \mathcal{Z} must have programmed any of the values $k||d$ or $k||i$ prior to the simulation.

Simulation with two malicious parties

It remains to prove security for the last case, where \mathcal{Z} corrupts both the sender and the receiver. The case of malicious sender and receiver does not guarantee any fairness and might not even terminate. Recall that we allow any malicious party to lose money if it does not follow the protocol execution, aborts, and declines the unfreezing request made by the ledger (cf. Section 5.2). A simple example of such a case is when \mathcal{S}^* does not trigger the **finalize** message when interacting with \mathcal{C} . This will result in his money staying blocked forever. But even if the standalone case of two dishonest parties does not make much sense for the proposed

applications, we still need to prove the indistinguishability of the real and hybrid world to guarantee security when composed with other protocols.

Claim 4. *There exists an efficient algorithm Sim^{SR} such that for all PPT environments \mathcal{Z} , that **corrupt both, sender and receiver** it holds that the execution of Π_{FAIRSWAP} in the $(\mathcal{C}, \mathcal{L}, \mathcal{H})$ -hybrid world in the presence of adversary \mathcal{A} is computationally indistinguishable from the ideal world execution of $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ with the ideal adversary Sim^{SR} .*

Proof. The simulation of the case with both malicious sender and receiver is a mixture of the two previous simulations. The simulator runs in the $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ world and needs to simulate the execution of Π_{FAIRSWAP} in the real world, using the inputs of \mathcal{S}^* and \mathcal{R}^* . Additionally, it needs to execute the functionality $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ on behalf of \mathcal{S}^* and \mathcal{R}^* to ensure the correct freezing of coins in \mathcal{L} . Since this case repeats the steps of the simulations with one malicious party, we will only give a high-level sketch of Sim^{SR} . Whenever \mathcal{R}^* or \mathcal{S}^* instruct \mathcal{L} not to pay out the coins, Sim^{SR} forwards this message to \mathcal{L} to enforce the identical behavior in the real and ideal world.

Simulator Sim^{SR}

1. Upon receiving message $(\text{init}, id, p, c, r_z, r_\phi)$ in the first step, Sim^{SR} simulates the execution of the hybrid functionality by sending $(\text{initialized}, p, c, r_\phi, r_z) = \mathcal{C}(\text{init}, id, p, c, r_z, r_\phi)$ to \mathcal{S}^* .

If \mathcal{S}^* did not send the message (init) , Sim^{SR} aborts the simulation.

Otherwise, if message $(\text{sell}, id, z, \phi, c)$ was also received from \mathcal{S}^* in the first step, Sim^{SR} sets $x^* = 1^{n \times \lambda}$ and sends $(\text{init}, id, \phi, p, x^*)$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$.

2. When Sim^{SR} receives (accept, id) by \mathcal{S}^* he sends $(\text{buy}, id, \mathcal{R})$ to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$. Additionally he runs $(\text{active}, id) = \mathcal{C}(\text{accept}, id)$. If no message from \mathcal{R}^* was received, Sim^{SR} terminates the simulation.
3. Upon receiving $(\text{reveal}, id, d, k)$ from \mathcal{S}^* in step 3, such that $\text{Open}(c, d, k) = 1$, Sim^{SR} triggers $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ to output of x^* to \mathcal{R} . Additionally Sim^{SR} runs $(\text{revealed}, id, d, k) \leftarrow \mathcal{C}(\text{reveal}, id, d, k)$ to simulate the execution of Π_{FAIRSWAP} .

If no message $(\text{reveal}, id, d, k)$ from \mathcal{S}^* is received, send (abort, id) to $\mathcal{F}_{\text{cfe}}^{\mathcal{L}}$ in the name of \mathcal{S}^* and simulate the refund in the hybrid world by running \mathcal{C} until it terminates.

4. Sim^{SR} waits to receive a message from \mathcal{R}^* .
- If the message is a valid complaint, $(\text{complain}, id, \pi)$ Sim^{SR} runs \mathcal{C} on the input of this message. If the judge contract accepts the complaint, he lets $\mathcal{F}_{cfe}^{\mathcal{L}}$ continue and terminates.
 - If the message is a false complaint or $(\text{finalize}, id)$ Sim^{SR} instead sends (abort, id) to $\mathcal{F}_{cfe}^{\mathcal{L}}$ and immediately trigger the following execution including the unfreezing of coins. Then Sim^{SR} terminates.
 - If instead \mathcal{R}^* does not send a message in the fourth step, Sim^{SR} waits for \mathcal{S}^* to send $(\text{finalize}, id)$ in step 5. If \mathcal{S}^* sends this message, Sim^{SR} triggers the further execution, including the payout of coins. Then Sim^{SR} terminates. If \mathcal{S}^* , on the other hand, does not send this message, Sim^{SR} also triggers the further execution of $\mathcal{F}_{cfe}^{\mathcal{L}}$, but when \mathcal{L} request the payout of the coins to \mathcal{S}^* , he refuses the request in the name of \mathcal{S}^* .

The simulator ensures that the coins are unfrozen correctly in \mathcal{L} by submitting a wrong file x^* to $\mathcal{F}_{cfe}^{\mathcal{L}}$. This allows him to wait for the payout phase and simulate the outcome of \mathcal{C} accordingly. If the receiver can produce a valid complaint, which will trigger \mathcal{C} to output the coins to \mathcal{R} , Sim^{SR} triggers the verification of the file, which will conclude that $\phi(x) \neq 1$. Otherwise, if \mathcal{R} does not produce a valid complaint, Sim^{SR} triggers the payout to \mathcal{S} in $\mathcal{F}_{cfe}^{\mathcal{L}}$ without checking if $\phi(x) = 1$ by sending (abort, id) . Now he waits for \mathcal{S}^* to send the finalize message. If this message is received within the expected time, he simulates the payout of coins. Otherwise, he needs to simulate that the coins are frozen in \mathcal{L} . He does this by refusing the payout of coins in \mathcal{L} when $\mathcal{F}_{cfe}^{\mathcal{L}}$ instructs it to send the payment of p coins to \mathcal{S}^* . This will block the execution, and the money is frozen. Since Sim^{SR} controls all outputs to the corrupted parties, it does not matter, that $\mathcal{F}_{cfe}^{\mathcal{L}}$ outputs a wrong file x , since Sim^{SR} could easily simulate the correct output from the internal execution of \mathcal{C} . This ensures that the execution of $\mathcal{F}_{cfe}^{\mathcal{L}}$ with Sim^{SR} is indistinguishable from the hybrid world execution to \mathcal{Z} . \square

5.6 Implementation and Performance

Now that we have proven the security of FAIRSWAP, we can take a closer look at the efficiency of the scheme. In this section, we show how to use our general

5 Moving Complex Computation Off-Chain

protocol for digital file sale and in more general use cases. We give performance indicators for our protocol and the costs of our implementation for the judge contract in Ethereum. While the runtime of FAIRSWAP is fixed to five steps, the length of a step depends on Δ rounds, as every step requires one interaction with the blockchain. This parameter is essential for the runtime and can be set by the parties individually, based on the application. Therefore, we analyze the protocol efficiency not on the runtime, but through the following aspects instead:

- **The cost of deployment for the judge contract.** This factor is largely influenced by the size of the instruction set and structure of ϕ , which the contract needs to support. Below we discuss how the judge implementation can be optimized to reduce this cost for our file sale example (when ϕ is a Merkle tree).
- **The cost for the optimistic protocol execution.** This cost is constant and independent of the size of ϕ and x .
- **The cost for the pessimistic protocol execution.** For simplicity, we will always consider the worst pessimistic case. The costs for a dispute, in this case, heavily depends on the size of the proof of misbehavior.
- **The size of the encoding.** Another way of measuring the protocol's efficiency is to consider the message complexities. The by far largest message of the protocol is the first one from \mathcal{S} to \mathcal{R} , which carries the encoding.

The numbers vary for different circuits ϕ , where circuits with small instruction alphabets Γ and fan-in ℓ are the most promising candidates. For such circuits, the overhead of the encoding is small, and the judge contract can run at low costs. To get exact numbers for these aspects, we implemented the judge smart contract for our file sale example. Before we present the findings, we discuss the circuit that we consider in more detail and give an overview of the implementation.

Efficient and Fair File Transfer. Our protocol can be used whenever two parties want to exchange data that is identified via its Merkle hash root h . The witness is the file $x = (x_1, \dots, x_n)$, which is split up into n parts, where each file chunk $x_i \in \{0, 1\}^\lambda$ is of some short length λ . The circuit ϕ checks if the Merkle hash root of this file equals some publicly given value h (i.e., $\phi(x) = 1$, iff $\text{Mtree}^{\mathcal{H}}(x) = h$). We assume that this value is made available via some public channel that the receiver trusts, e.g., a webpage or file sharing log. The instruction alphabet of ϕ

5 Moving Complex Computation Off-Chain

consists of the operations $H(x, y)$ and $eq(x, h)$, where H is the Hash function used for the Merkle tree and $eq(x, y)$ is a function that outputs 1 if $x = y$. Figure 5.9 shows such a circuit for a file with $n = 8$ elements of size λ .

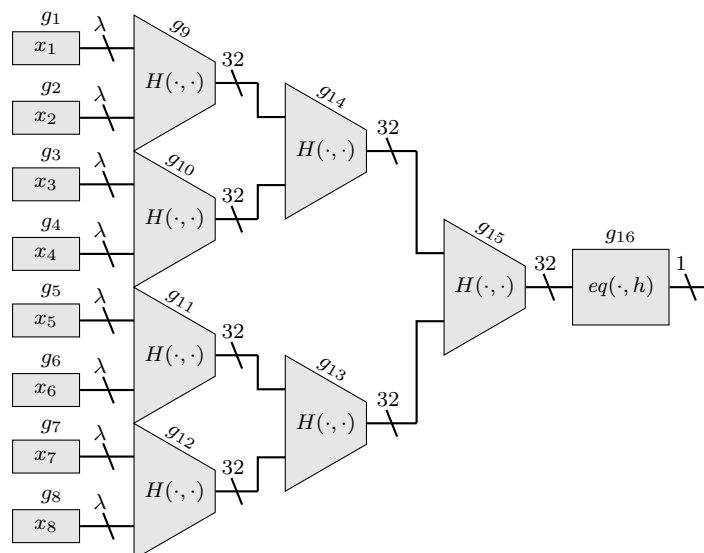


Figure 5.9: Merkle tree circuit for exchange of $x = (x_1, \dots, x_8)$

To benchmark the runtime and execution costs of our protocol, we implemented the protocol for the file sale application⁹ using the file sale circuit (cf. Figure 5.9). A nice property of this circuit is the small size of the instruction alphabet ($|\Gamma| = 2$), and the small fan-in of operations ($\ell = 2$). This allows us to provide a highly efficient smart contract implementation for this particular use case. The advantage of the small instruction alphabet is that the contract can derive the operation of gate g_i from the index i (indeed, there is only one operation in the entire circuit ϕ except for the very last instruction). This allows us to implement the verification without committing, sending, and verifying ϕ_i . Additionally, for the special case of a Merkle tree circuit, we have that the input to all gates (i.e., hash function evaluations) are natural siblings in the encoding in z . This means that in the concise proof of misbehavior to verify the correct evaluation of one hash functions on two inputs, we only need one (slightly modified) Merkle proof verification, which verifies both input values in one step. Thus, the proof π only includes two

⁹The source code of the solidity contract can be found at github.com/1EthDev/FairSwap

5 Moving Complex Computation Off-Chain

input values of at most length λ , one output hash of length μ , and two Merkle proofs – one for the two input elements and one for the output of the gate. In our implementation, the users have the ability to change the parameters of the protocol, namely the number of file chunks n , which directly relates to two other parameters in our application: the length of each file chunk $|x_i| = \lambda$ and the depth of the Merkle tree δ (again we assume a full tree for simplicity). We can observe the following relation of the parameters:

$$\lambda = \frac{|x|}{n} = \frac{|x|}{2^\delta}$$

The hash function optimized in the Ethereum virtual machine language is `keccak256`, which outputs hashes of size $\mu = 32$ bytes. Since the instruction set of Solidity is currently limited but provides a relatively cheap (in gas costs) and easy hashing, we use this hash function to implement our encoding scheme. Since the judge contract needs the possibility to extract each element $z_i \in z$ without knowledge of the whole vector z we use a variant of the plain counter mode for symmetric encryption, for which `keccak256` is evaluated on input of a key k and index i , and the ciphertext is the bitwise `XOR` of the plaintext with this hash output taking as input the key and the current counter. From the construction of the encoding scheme, it follows that the file chunk length λ should be a multiple of 32 bytes to allow efficient encoding and extraction.

The judge contract implementations offers four different options for \mathcal{R} to call during the `payout` phase. The function `nocomplain` allows \mathcal{R} to accept the file transfer and directly send p coins to \mathcal{S} , the `complainAboutRoot` function is used whenever \mathcal{R} complains about a false output of the circuit, namely that $z_m \neq h$. The functions `complainAboutLeaf` and `complainAboutNode` allow \mathcal{R} to complain about the computation of two input gates g_i, g_{i+1} , $i \in 1 \dots, n$ or the computation of some other gates g_i, g_{j+1} where $n \leq j < m$ respectively. The reason for these different complain functions is that each of them requires a differently sized input.

Transaction Fee Evaluation. In the first round, the sender deploys the *FairSwap* contract, including the Ethereum addresses of \mathcal{S} and \mathcal{R} , the price value p , the commitment c and the roots r_ϕ and r_z . The main gas costs result from this deployment, which costs roughly 1050000 gas, which for a gas price of 3 GWei translates to 0.00315 Ether or 0.51 euros.

The price for the execution of the functions `deploy`, `accept`, `reveal`, `refund` and `no complain` stays almost constant for different parameters, but the cost of the

5 Moving Complex Computation Off-Chain

complain function varies highly, depending on the kind of complaint, the choice of the file chunk size λ and the Merkle depth δ . The data, which needs to be sent to the blockchain (as part of π), increases with the size of λ . Figure 5.10 shows the costs for optimistic (green) and pessimistic (red) execution costs for different file chunk lengths for a file of one GByte size. Optimistic means, that the protocol continues until the **payout** phase, where \mathcal{R} accepts the encoding without complaint, whereas pessimistic means, that \mathcal{R} complains to the judge contract about the wrong computation of some input. The costs of the complaint originate from the length of the concise proof of misbehavior π , which needs to be sent to the contract and evaluated on-chain:

$$\begin{aligned} |\pi| &= |z_{in1}| + |z_{in1}| + |z_{out}| + |\rho_{in}| + |\rho_{out}| \\ &\leq 2\mu \times \lambda + \mu + 2\delta \times \mu \end{aligned}$$

Figure 5.10 illustrates that even for very large file chunk sizes, the costs for optimistic execution is close to constant around 0.56 euros, where the cost for pessimistic execution increases exponentially in the length of the file chunks. We highlight that using different cryptocurrencies can decrease the price for execution even further. In Ethereum classic¹⁰, a well known fork of the Ethereum blockchain, the cost for optimistic execution is only fractions of cents. The heavy computation of the protocol is executed in round 1 and 3, by both the sender and the receiver in the two algorithms $\text{Extract}^{\mathcal{H}}$ and $\text{Encode}^{\mathcal{H}}$. We will only take a closer look at the performance of the sender, since the receiver will perform almost identical computations only in reverse order. To encode file $x = (x_1, \dots, x_n)$, \mathcal{S} needs to first generate $M = \text{Mtree}^{\mathcal{H}}(x)$ and store all intermediate hashes. The result is $n - 1$ elements of size μ . Next, he encodes each file chunk (which requires $n \times \lambda$ hashes in total) and each hash from the Merkle tree M ($n - 1$ hashes). It remains to compute the Merkle root $r_z = \text{Mtree}^{\mathcal{H}}(z_1, \dots, z_m)$ of the combined encoding $Z = z_1, \dots, z_m$, for which he needs to hash $2m$ Elements. Therefore we get the following estimates:

$$\begin{aligned} |z| &= |x| + (n - 1)32 \text{ Bytes} = n \times \lambda \times 32 \text{ Bytes} \\ \text{Runtime of } \text{Extract}^{\mathcal{H}} &= n \times \lambda \times \mathcal{O}(\mathcal{H}) \end{aligned}$$

The size of z therefore can be used as an indicator for the performance of the algorithm $\text{Extract}^{\mathcal{H}}$ and additionally affects the communication complexity, since

¹⁰<https://ethereumclassic.github.io/>

5 Moving Complex Computation Off-Chain

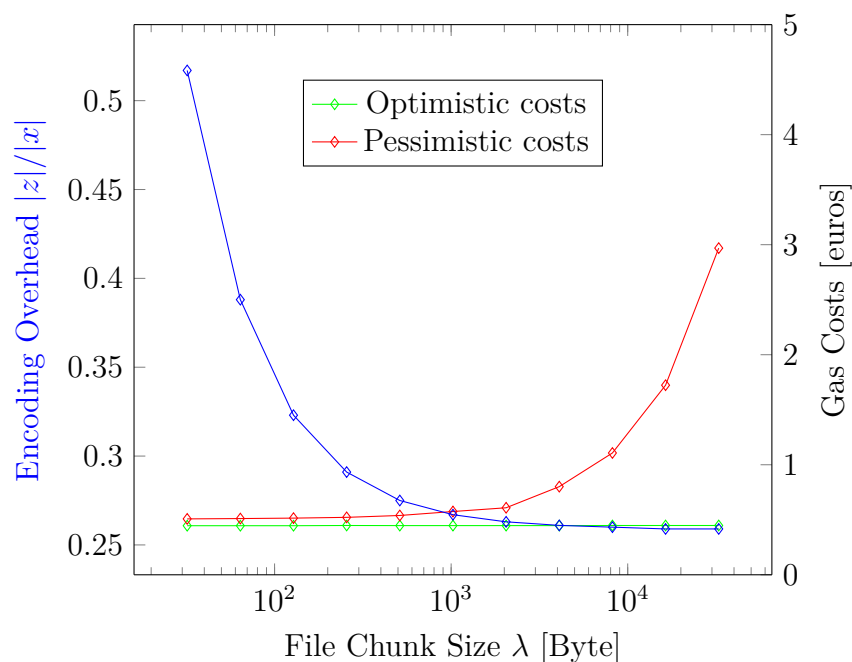


Figure 5.10: Costs and encoding size for different values of λ

it needs to be transferred in the first protocol message which is the longest message in the protocol. We did not optimize the implementation of the algorithms $\text{Encode}^{\mathcal{H}}$, $\text{Extract}^{\mathcal{H}}$ for runtime, but it shall be noted that we reach an encoding throughput of approximately 2 MB per second in a straightforward node.js implementation running on a single core of a 2.67 GHz Intel©Core i7 CPU with 8 GB of RAM.

For most use-cases the execution cost will dominate the costs for extraction and message sending. This indicates that the performance of the protocol is often optimal for small λ . Figure 5.10 illustrates this trade-off between the costs of the protocol and its performance, measured in the overhead of encoding size in comparison to file length $\frac{|z|}{|x|}$. The protocol can be executed in 4 rounds, where each round requires sending a message to the blockchain. The round ends when the message is accepted by the miners and included in a block. Cryptocurrencies ensure that a correct message (with sufficient fees) is eventually included in the blockchain, but this process might take some time. We denote this maximal round duration with Δ . Therefore the judge contract will have timeouts Δ to measure whether some message has been sent or not. The exact value of these parameters is

chosen by the parties and depends on the congestion of the blockchain, the number of fees they are willing to pay, their availability, and the number of blocks they require to succeed a transaction in order to consider it valid. We note that the minimum duration of the protocol is four blockchain rounds, which in Ethereum can be executed in only a few minutes, as long both parties agree.

5.7 Discussion and Extension

In the previous chapters, we have presented the FAIRSWAP protocol and analyzed its security and efficiency. In the optimistic case, the protocol is executed in 5 rounds and costs around 0.56 euros. In this section, we discuss how to apply the system to solve free-riding in distributed systems. We additionally present different extensions to the protocol. In Subsections 5.7.2 and 5.7.3, we discuss how the communication overhead and transaction fees in the optimistic case can be reduced by applying the ideas from the follow-up works [73] and [183]. In Subsection 5.7.4, we discuss how incentives can be added to the protocol and how FAIRSWAP can be executed in state channels.

5.7.1 Countermeasure against Free-Riding

We discussed the file sale application of FAIRSWAP in detail in the previous section. Here we want to highlight how FAIRSWAP can provide an elegant solution to the free-riding problem in distributed file-sharing systems. The free-riding problem states that the overall network suffers if enough peers only exploit the service without sharing content themselves. Surveys like [164, 166] show that free-riding is a common problem in decentralized systems whenever creating identities is cheap, and users can dynamically join and leave the system. In [3], researchers found that at one point, 75% of users of the popular platform Gnutella were free-riders. Some incentive mechanisms have been proposed to make free-riding less attractive [107], e.g., by introducing payments and let users pay small fees to the senders of files. However, such approaches do not address the case when a malicious user offers content that is incorrect (i.e., it does not belong to the file that the receiver intends to download). A natural solution to the free-rider problem is to use cryptocurrencies that support smart contracts since they provide a decentralized trust platform that handles payments. One possible solution already discussed in the introduction is to use ZKCP, but this only works well for small inputs as otherwise, the users would suffer from huge computational penalties. FAIRSWAP however, solves the

fairness problems of digital file exchange for much lower costs and results only in small overheads in terms of communication load.

5.7.2 Interactive Dispute

In an extension to the original FAIRSWAP protocol called OPTISWAP [73], we propose an interactive dispute process. As a result, we can reduce the communication size, in particular the first message sent from \mathcal{S} to \mathcal{R} , and propose a fairer fee distribution. The high-level idea of the OPTISWAP protocol is as follows: In the first message, \mathcal{S} sends only an encoding of x , not of the transcript of the evaluation of $\phi(x)$ to \mathcal{R} . Otherwise, the protocol proceeds as in the FAIRSWAP protocol. In case there is a dispute, the receiver \mathcal{R} does not have enough information to build a proof of misbehavior right away. Instead, he starts the interactive dispute procedure, which allows him to learn the missing elements from \mathcal{S} . Both parties run a challenge-response procedure via the contract, in which \mathcal{R} requests the results of single gates. If \mathcal{S} does not respond (correctly) in time, the contract will notice this misbehavior and assign the coins to \mathcal{R} . Otherwise, the challenge-response process will be finished within an a-priori fixed number of challenge-response pairs, and \mathcal{R} can prove the misbehavior of a cheating \mathcal{S} .

This dispute resolution sub-protocol is only used in the pessimistic case and does not add to the execution fees of the optimistic case, while drastically reducing the overhead as observed in 5.6. In particular, the size of the message in FAIRSWAP depends on the circuit and the witness size, while the message size in OPTISWAP is independent of the circuit size. The paper includes a prototype implementation and full security proof in the UC model of the interactive dispute protocol. The authors of [183] implemented the fair file sale application in their framework and managed to reduce the costs by 22%.

5.7.3 Splitting Escrow and Judge Function

In the optimistic case, the main cost for running fair file sale consists of the deployment costs, which make 91% of the overall costs. In the original publication of FAIRSWAP [67], we proposed to re-use the contract whenever the same circuit is re-used, between the same parties. The contract needs to be slightly modified such that it can store money for repeated exchanges and updates the internal balance after every execution. This approach increases the deployment costs slightly (around 0.06 euros), the cost amortizes when the contract is re-used. While

5 Moving Complex Computation Off-Chain

this approach can help to amortize the deployment costs, it only helps in the repeated exchange between two parties. The authors of [183] propose a more generic method to improve deployment costs. They distinguish two parts of the contract, the *mediator* function and the *verifier* function, as follows:

Tasks of mediator

- Store address of verifier contract
- Store coins from \mathcal{R}
- Store exchange data $(id, \phi, p, c, r_\phi, r_z)$
- Store opening (d, k) from \mathcal{S}
- Enforce timeouts
- Payout the stored coins

Task of verifier

- Stores the logic to evaluate function $\text{Judge}^{\mathcal{H}}$
- Receive proof of misbehavior
- Judge if \mathcal{S} was honest by evaluating $\text{Judge}^{\mathcal{H}}$
- Report the result to escrow contract

The contract now has two parts, the escrow part which stores the result of the contract protocol execution and the judge part which verifies disputes. We can split up the contract into two parts like this, as the judge is only needed in the pessimistic case. Splitting up the contract allows users to re-use deployed verification contracts for judging the same circuit evaluation. Additionally, the verification contract can also be deployed on demand only during dispute with the deterministic `create2` opcode (cf. Section 3.2.1).

	Costs	in Gas	in ETH	in Euro
Deployment Costs	FAIRSWAP	1050000	0.00315	0.51
	OptiSwap [73]	2273398	0.006820194	1.11
	SmartJudge [183]	1947000	0.005841	0.95
Costs for the Optimistic Case	FAIRSWAP	1153303	0.00346	0.56
	OptiSwap [73]	101307	0.000303921	0.05
	SmartJudge [183]	143000	0.000429	0.07

Table 5.1: Gas cost comparison between FairSwap, SmartJudge, and OptiSwap. Numbers taken from related work (with exchange rates from Section 3.2.1).

In “OptiSwap: Fast Optimistic Fair Exchange” [73] the gas costs of the FAIR-SWAP, OPTISWAP and SmartJudge protocols were evaluated and compared. The

result is presented in Table 5.1. The results show that SmartJudge managed to reduce the deployment costs of FAIRSWAP and the OPTISWAP protocol lowered the execution costs in the optimistic case (additionally to the reduced overhead due to the interactive dispute). As both approaches use different ideas which do not exclude each other, they can also be combined to minimize both the deployment and execution costs. We note, however, that both techniques are more expensive than FAIRSWAP in the pessimistic case (as discussed above). Therefore, our next extension discusses how to set clear incentives for both parties to avoid dispute.

5.7.4 Setting Financial Incentives for Honest Behavior

We have conducted a formal proof that FAIRSWAP is secure against malicious behavior, which guarantees that no sender \mathcal{S} can get the coins without revealing the witness x to \mathcal{R} and no \mathcal{R} can get the witness without paying the honest sender \mathcal{S} . But we did not prevent that the parties can harm each other in another way. In particular, the sender can force \mathcal{R} to lock his coins for 4Δ rounds without even knowing x (cf. collateral costs Section 3.2.2). We cannot prevent this behavior because \mathcal{R} cannot see the witness before the third round. But we can incentivise \mathcal{S} to behave honestly, to penalize this behavior. This is possible because the contract can always correctly attribute the fault in the pessimistic case. In order to be able to penalize \mathcal{S} , he first needs to lock a penalty deposit of $p_{\mathcal{S}}$ coins in the contract in the first round. If he behaves correctly (i.e., follows the protocol description and provides a correct witness), the contract sends the payment and his own penalty deposit back to him. In case \mathcal{S} cheats or aborts, the money is instead sent to \mathcal{R} as compensation¹¹. A downside of this incentive mechanism is that it increases the collateral costs for honest senders.

Another potential attack vector of the protocol is grieving through transaction fees (cf. Section 3.2.2). So far, we did analyze how much fees need to be paid for the execution of the protocol, but we did not discuss their unequal distribution between the parties. In FAIRSWAP, the deployment fees are carried only by \mathcal{S} , and in case of dispute, \mathcal{R} has to pay all fees for sending the proof of misbehavior and evaluating Judge^H. The deployment costs can be covered just as well from \mathcal{R} , but we recommend to use one of the previous approaches to reduce the deployment costs overall. An interesting solution to achieve fair fee distribution and set incentives for honest behavior was introduced in [73]. Here we analyze how the fees of (interactive) dispute can be shifted completely to the misbehaving party.

¹¹The money could also remain locked forever in the contract with the same result.

Again, we use the guaranteed fault attribution to identify the malicious party and use its deposit to compensate the honest party. In particular, we will distinguish three possible cases:

1. When both parties behave honestly, the exchange proceeds without any additional fees or deposits.
2. When \mathcal{S} is honest and \mathcal{R} forces a dispute case, \mathcal{S} might need to temporarily lock a deposit, but he is guaranteed to get it back and get compensated for any fees that he paid during the process.
3. When \mathcal{R} is honest and needs to dispute about a false witness, he might need to temporarily lock a deposit, but he is guaranteed to get it back and get compensated for any fees that he paid during the process.

5.7.5 FairSwap in Channels

To minimize gas costs and confirmation time for repeated execution, we propose to run the judge contract described above off-chain in a state-channel (cf. Section 4.7). State channels are an extension of payment channels and allow users to execute arbitrary smart contracts off-chain without requiring interaction with the blockchain. Constructions for state channels have been proposed in earlier works, e.g., in [148, 71]. In our basic system, \mathcal{S} and \mathcal{R} open a state channel for the (repeated) execution of FAIRSWAP. If the parties want to execute multiple fair exchanges, they freeze a sufficient number of coins in the channel. Now the users can run multiple fair exchange executions without sending blockchain transactions, which drastically decreases the fees and execution times. If however, at some point one of the parties starts to behave maliciously (e.g., a sender does not provide the secret key for the i -th repetition of the protocol) the parties can always execute the contract for this repetition on-chain and settle their disagreement in a fair manner. Payment networks are an even more interesting tool here, when the goal is to connect many potential buyers and sellers – for example in the distributed file sharing use-case. With only a single on-chain setup a user can connect to many other users off-chain. Virtual state channels [71] can be utilized to create and close new state channels for every new connection off-chain. As long as the user and its connection to the network behave honestly in the channel and the FAIRSWAP protocol, no on-chain interaction is necessary for this user. Even if the counterparty of the virtual connection starts to misbehave, the indirect dispute

5 Moving Complex Computation Off-Chain

of [71] ensures that only the malicious party and its direct on-chain counterpart dispute on-chain and carry the on-chain fees.

6 Off-Chain Smart Contracts on Bitcoin

Summary. So far, in this thesis, we have examined how smart contracts can be used to build complex applications on blockchain technology. But many legacy cryptocurrencies like Bitcoin do not support smart contracts, but only very simple transactions. In this chapter, we will explore a different approach to taking complex contracts off-chain. In contrast to the previous chapter, we will consider arbitrary contracts with more than two players. Instead of executing the contracts on-chain or locally at the parties, we will outsource this task to TEEs.

In this chapter, we propose and analyze a TEE based scaling protocol, which is based on the publication “FastKitten: Practical Smart Contracts on Bitcoin” [59] — presented at the 2019 USENIX Security Symposium. In this work, we present FASTKITTEN , an efficient protocol for executing generic smart contracts off-chain at low costs over cryptocurrencies with reduced scripting capabilities. We execute contracts off-chain in TEEs operated by an untrusted party and ensure through penalties that malicious parties that prevent the contract execution will always be punished. This setup allows the fast and cheap execution of smart contracts, that can be arbitrarily complex and are independent of the underlying scripting language. We show that we only require very simple transactions on the underlying blockchain and that this system can be deployed on top of Bitcoin. We formally prove strong security properties and show the feasibility of the construction based on a prototype implementation.

6.1 Overview

Since the rise of Bitcoin, countless new cryptocurrencies have been launched to address some of the shortcomings of Nakamoto’s original proposal. But despite these developments, Bitcoin still remains by far the most popular and intensively studied cryptocurrency, with its current market capitalization that accounts for more than 50% of the total cryptocurrency market size [55]. A particular important

shortcoming of Bitcoin is its limited scripting capabilities, which prevents the support of smart contracts like the ones we modeled in the previous chapters. The reduced scripting language is part of the design rationale of Bitcoin, and will most likely not be changed. But limitations do not make it impossible to design smart contracts, as many previous works have built protocols secured by Bitcoin before [5, 123, 124, 121]. But while it is technically possible, it requires significant financial costs due to complex transactions and high collateral.

Another option for running more complex smart contracts is Ethereum, where the contract execution is directly integrated into the consensus mechanics. But this forces every miner to execute and verify all contract calls which they are paid for in gas. Additionally, the gas limit sets bounds on the number of instructions that can be evaluated per block. Both factors limit the applications for smart contracts. Finally, many applications for smart contracts require confidentiality, which is currently not supported by either Bitcoin or Ethereum.

In this chapter, which is based on [59], we propose the FASTKITTEN protocol, which leverages TEEs to run arbitrarily complex smart contracts at low costs on top of cryptocurrencies as Bitcoin. We use a TEE to evaluate the contract inside an enclave, shielding it from potentially malicious users, including the operator of the TEE. Moving the contract execution into the secure enclave guarantees correct and private evaluation of the smart contract even if it is not running on the blockchain and verified by the decentralized network. This approach circumvents the efficiency shortcoming of cryptocurrencies like Ethereum, where contracts have to be executed in parallel by thousands of users.

Again, we will build an optimistic protocol, which only requires blockchain interaction during the setup and closing phase when all parties behave maliciously. This case will be very efficient even for complex smart contracts, which is executed in many iterations (we say rounds) by many players. We will also show that in the pessimistic case, the protocol is still secure, which means that the smart contract will always evaluate correctly based on the correct inputs of the parties. While we cannot prevent that some parties, mainly the operator of the TEE, has the power to stop the contract execution, we can guarantee financial fairness. This property guarantees that misbehavior such as stopping the contract execution will always be punished and honest parties will be compensated with at least the amount they stored in the smart contract.

Furthermore, the protocol is blockchain agnostic, but a reference implementation shows the feasibility and evaluates the costs for Bitcoin [59]. We emphasize that FASTKITTEN requires only a single TEE that can be owned either by one

of the participants or by an external service provider, which we call the *operator*. In addition, smart contracts running in the FASTKITTEN execution framework support private state and secure inputs, and thus, offer even more powerful contracts than Ethereum. We will also discuss how the protocol can be extended, for example, by building smart contracts that can operate different cryptocurrencies.

6.1.1 Intuition and Design Ideas

We consider the following setup where a fixed set of n users $\mathcal{P}_1, \dots, \mathcal{P}_n$ want to execute an arbitrary complex smart contract C with deposits of c_1, \dots, c_n coins. During the initialization phase, the contract receives these coins from the parties and some initial inputs. Next, it runs for m reactive rounds, and in each round, the contract can receive additional inputs from the parties \mathcal{P}_i and produces a round output. Finally, after the m -th round is completed, the contract terminates. If the contract proceeds to the final step, the last output specifies the coin distribution of the locked coins. The protocol ensures that in the optimistic case, this output is enforced. In the pessimistic case, it guarantees that, if the contract is not executed until the end, all parties get their initial coins back, and cheating parties do not. In the optimistic case, FASTKITTEN runs very fast, since the entire contract is evaluated off-chain and only requires the blockchain during the coin (un-)locking in the contract initialization and finalization.

We build the FASTKITTEN protocol over a decentralized cryptocurrency that only supports very simple scripts. Concretely, we require (i) simple transactions that send coins to public keys, (ii) transactions that can store data, and (iii) *time-locked transactions*, that are only processed and integrated into the blockchain after a specified time has passed. We emphasize that these are very mild requirements on the underlying cryptocurrency that, for instance, are satisfied by the most prominent cryptocurrency Bitcoin (cf. Section 3.1).¹ FASTKITTEN leverages these properties together with the power of trusted execution environments to provide an efficient general-purpose smart contract execution platform.

The key feature of FASTKITTEN, its very low execution cost, and high performance, is achieved by running the contract within a single TEE like Intel’s Software Guard Extensions (SGX) [142, 99, 4] or ARM TrustZone [7]. Such TEEs are part of a processor and allow the creation of protected *enclaves* — computation environments that strictly isolate the state and memory of a specific application,

¹Bitcoin transactions can store up to 97 KB of data [139]; multiple transactions can be used for bigger payloads.

6 Off-Chain Smart Contracts on Bitcoin

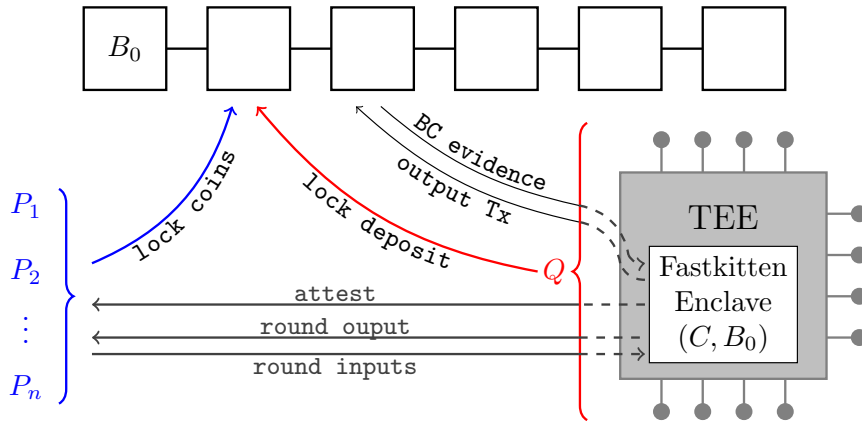


Figure 6.1: Setup and communication of FASTKITTEN protocol.

thus allowing confidential and correct computations. The owner of the machine of the TEE is a party that we call the *operator* Q . In practice, this operator will either be one of the users or a designated service provider that takes a small fee for this service. Outsourcing the access to the TEE implies that ownership of a TEE is not required to run a contract, making the system easily accessible for everyone. While the TEE itself is trusted and the parties can attest that the TEE is running the correct code, the operator is not trusted. We require that he also deposits a (large) security deposit during the runtime of the contract. He will lose this deposit if he stops running the TEE or relaying the correct messages. If he behaves honestly, he will get all deposited coins (plus his payment). To enable secure off-chain contract execution, our architecture builds on existing TEEs, which are widely available through commercial off-the-shelf hardware.

As depicted in Figure 6.1, the FASTKITTEN enclave is responsible for executing contract C and is initialized with the initial blockchain checkpoint B_0 . The enclave is executed by the TEE, which is in the control of the operator Q . The untrusted host process of Q takes care of setting up the enclave and handles the communication to the other participants and the blockchain. While this means that Q has complete control over relaying the communication, the influence of a malicious operator on a running enclave is limited: he can interrupt enclave execution, but not tamper with it. Further, the enclave will sign all code and data as part of its *attestation* towards parties, so they can verify the correctness of the setup before locking their coins. The contracts are loaded into the FASTKITTEN enclave during the initialization of our protocol by the underlying host process, and participants can verify that contracts are loaded correctly. To support arbitrary contract func-

tionality, the FASTKITTEN enclave includes additional verification and protocol functionality to generate and verify transactions, and pass data from the contract to the host.

Our protocol then proceeds in three phases, which we call *setup phase*, *round computation*, and *finalization phase*. During the setup phase, the contract is loaded into the enclave, and all parties $\mathcal{P}_1, \dots, \mathcal{P}_n$ can verify (attest) that this step was completed correctly. Then, the operator and all parties block their coins for the contract execution. If any party aborts in this phase, the money is refunded to all parties that deposited money, and the protocol stops. Otherwise, all parties receive a time-locked penalty transaction, needed in case \mathcal{Q} aborts the protocol. Afterwards, the round computation phase starts, in which \mathcal{Q} sends the previous round’s output to all parties. If a party \mathcal{P}_i receives such an output, which is correctly signed by the enclave, it signs and sends the input for the following round to \mathcal{Q} . If all parties behave honestly, \mathcal{Q} will forward the received round inputs to the enclave, which computes the outputs for the next round.

The last phase of the protocol is the payout phase. In this phase, the enclave returns the output transaction, which distributes the coins according to the terminated contract. In case of a protocol abort, the coins initially put by the users will be refunded to all honest parties. If any party was caught cheating, this party would not receive back its coins. This means the money will stay in control of the enclave and will never be spent.

Design Challenges of FastKitten

Leveraging TEEs for building a general-purpose contract execution platform requires us to resolve the following main challenges.

Protection against a malicious operator. The operator runs the TEE and hence controls all interaction with its environment (e.g., with the other protocol participants or the blockchain). Thus, the operator can abort the execution of the TEE, delay and change inputs, or drop any ingoing or outgoing message. To protect honest users from such an operator, the enclave program running inside the TEE must identify such malicious behavior and punish the operator. In particular, we require that even if the TEE execution is aborted, all parties must be able to get their coins refunded eventually. To achieve this, we let the operator create a so-called *penalty transaction*: this transaction time-locks the deposit coins of the operator, which in case of misbehavior can be used to refund the users and punish

the operator.

Designing such a scheme for punishment is highly non-trivial since it requires the protocol to attribute faults. In particular, it must be clear whether it was the fault of the operator \mathcal{Q} or a party \mathcal{P}_i when an expected round input from \mathcal{P}_i does not arrive. From the point of view of the enclave that runs the contract, it is not clear whether the operator did not forward a message to the enclave or party \mathcal{P}_i did not send the required message to the operator. Since it is clear that at least one of the parties did not behave honestly, fault attribution is necessary, and we leverage the blockchain to carry out a challenge-response mechanism: The TEE will ask the operator to challenge \mathcal{P}_i via the blockchain. The operator can then either deliver a proof that he challenged \mathcal{P}_i via the blockchain but did not receive a response (in time), in which case \mathcal{P}_i will get punished²; or the operator receives \mathcal{P}_i 's input and can continue with the protocol. Note that this is only required in the pessimistic case, and this costly and time-consuming mechanism will never occur when both the party and the operator are behaving honestly. Whenever a part of the blockchain has been successfully verified by the enclave, it will store the last block as the new checkpoint, simplifying future verifications.

Verification of blockchain evidence. We do not assume that the TEE is an active node of the blockchain, i.e., it does not need to verify every single block or validate the correctness of the transactions, as this would lead to a great computational effort.³ Instead the operator feeds blocks and transactions, which we call *blockchain evidence* – to the TEE only if necessary. To ensure that a malicious operator cannot forge this blockchain evidence, we need to design a secure blockchain validation algorithm that can efficiently be executed inside a TEE. We achieve this by simplifying the verification process typically carried out by full blockchain nodes by using a checkpoint block (or genesis block) to serve as the initial starting point for verification. This is secure if all participating parties agree on this genesis block. To further speed up the computation inside the TEE, we only validate the minimum amount of information necessary. This includes the correctness of block headers and protocol transactions while ignoring all other transactions. In order

²Alternatively, we could allow the operator to spend the challenge transaction after a timeout has passed. While this would result in easier verification for the TEE, the operator would need to publish an additional transaction, increasing both fees and the overall time for the challenge-response phase.

³In fact, we do not even assume that the TEE has access to the current time, but we do assume that it has a strictly monotonic counter, which is increased with every activation of the enclave.

to be secure against forks and double-spending, a block is only considered by the TEE, if it has been confirmed by at least a specified number⁴ of blocks.

Preventing denial of service attacks. Complex smart contracts may take a very long time to complete, and in the worst case, there is no guarantee that they will terminate. Hence, a malicious party may carry out a denial-of-service attack against the contract execution platform, where the platform is asked to execute a contract that never halts. It is well known that determining whether a program terminates is undecidable. Hence, general-purpose contract platforms, such as Ethereum, mitigate this risk by letting users pay via fees for every step of the contract execution. This effectively limits the amount of computation that can be carried out by the contract. Since FASTKITTEN allows multiple parties to provide input to the contract in the same round, it might be impossible to decide which party (parties) caused the denial of service and should pay the fee. To this end, FASTKITTEN protects against such denial-of-service attacks using a timeout mechanism. As all users of the system (including the operator) have to agree on the contract to be executed, we assume that this agreement includes a limit on the maximum amount of execution steps that can be performed inside the enclave per one execution round. See Section 6.6.1 for more details.

6.1.2 Related Work

In this section, we will focus on related work, which considers smart contract execution on Bitcoin. We separately discuss multi-party computation based smart contracts and solutions using a TEE. We also provide a discussion on how Ethereum based solutions compare to FASTKITTEN.

Multi-party computation for smart contracts An interesting direction to realize complex contracts over Bitcoin is to use so-called multi-party computation with penalties [123, 124, 121]. Similar to FASTKITTEN, these works allow secure m -round contract execution, but they rely on the claim-or-refund functionality [123]. Such functionality can be instantiated over Bitcoin, and hence these works illustrate the feasibility of generic contracts over Bitcoin. Unfortunately, solutions supporting generic contracts require complex (and expensive) Bitcoin transactions and high collateral locked by the parties, which makes them impractical for most use-cases. Concretely, in all generic n -party contract solutions we are aware of,

⁴This number can be adapted to fit the application.

each party needs to lock $\mathcal{O}(nm)$ coins, which overall results in $\mathcal{O}(n^2m)$ of locked collateral. In contrast, the total collateral in FASTKITTEN is $\mathcal{O}(n)$, see column *collateral* in Table 6.1. It has been shown that for specific applications, concretely, a multi-party lottery, significant improvements in the required collateral are possible when using MPC-based solutions [147]. This, however, comes at the cost of an inefficient setup phase, communication complexity of order $\mathcal{O}(2^n)$, and $\mathcal{O}(\log n)$ on-chain transactions for the execution phase. Let us stress that the approach used in [147] cannot be applied to generic contracts. Overall, while MPC-based contracts are an interesting direction for further research, we emphasize that these systems are currently far from providing a truly practical general-purpose platform for contract execution over Bitcoin, which is the main goal of FASTKITTEN.

TEEs for blockchains There has recently been a large body of work on using TEEs to improve certain applications on blockchains [191, 192, 22, 131, 179]. A prominent example is Teechain [131], which enables off-chain payment channel systems over Bitcoin. However, like most of these prior works, Teechain does not use the TEE for smart contract execution. Some other works, including Hawk [120] and the “The Ring of Gyges: Investigating the Future of Criminal Smart Contracts” [104], propose privacy-preserving off-chain contract execution using TEEs but do not work over Bitcoin.

Probably most related to our work are [52, 31, 106], which propose blockchain agnostic systems for private off-chain function execution using TEEs. Despite the conceptual similarities of these works and FastKitten, the goals are orthogonal. The goal of the above-mentioned works is to move heavy computation off the chain in order to reduce the cost of executing complex contract functions. However, they do not aim to reduce the communication load on the blockchain. In fact, the communication complexity is often increased compared to the naive on-chain execution of the contracts. In contrast, FASTKITTEN aims to minimize the on-chain communication (especially for multi-round applications) and hence can be viewed as a full-fledged blockchain scaling solution. The works [52, 31, 106] consider clients (contract parties) and computing nodes which have a similar task as FASTKITTEN’s TEE operator since they also execute contracts inside a TEE. In contrast to FASTKITTEN, they send the encryption of the resulting contract state to the blockchain after every function call. If a client requests another function call, a selected computing node takes the state from the blockchain, decrypts it inside its enclave, and performs the contract execution. This implies that reactive multi-round contracts are very costly even in the standard case when

all participating parties are honest (cf. column *Minimal # TX* in Table 1). All works [52, 31, 106] rely on multiple TEEs to guarantee service availability as long as at least one TEE is controlled by an honest computing node. Even though FASTKITTEN only relies on a single TEE operator, we discuss in Section 6.7.5 how fault tolerance can be integrated into the system in a straightforward way. A joint goal of all systems is to provide state privacy of the contracts. Since one of the main goals of FASTKITTEN is to provide a scalability solution of multi-round applications, it incentivizes parties to minimize the blockchain interaction as much as possible. Fair distribution of coins is guaranteed through penalizing malicious parties. Since this is not the focus of [52, 31, 106], fair coin distribution is not discussed. For example, Ekiden does not even model or discuss the handling of coins. It is not straightforward to add this feature to their model since the contract state is encrypted, and hence the money cannot be unlocked automatically on-chain. The works [52, 31, 106] are independent projects that have similar goals and approaches. The main difference between Private Data Object [31], compared to Ekiden [52] is the way in which keys are distributed among TEEs. In [31], the owner of a contract can decide himself which computing nodes get access to the decryption key needed for the contract state decryption. This is in contrast to Ekiden, where all computing nodes have access rights by default. On the other hand, Ekiden aims to achieve forward secrecy even if a small fraction of TEEs gets corrupted via, e.g., a side-channel attack. Their strategy is to secret-share a long-term secret key between the TEEs and use it to generate a short-term secret key every epoch. Hence, an attacker learning the short-term key can only decrypt state from the current epoch⁵. The work of [106] provides a more generic construction for using blockchain to achieve statefulness and connectivity of TEEs compared to [52, 31]. In addition, it provides a formal model, a rigorous security analysis, and discusses multiple applications, like private smart contracts or fairness in multi-party computation.

Ethereum based solutions. A large body of research aims at reducing the cost and overheads of smart contract evaluation on-chain. Examples for these systems include state channels [148, 69], Arbitrum [105], Plasma [160] and [177]. However, these works do require an Ethereum like cryptocurrency and cannot be integrated into popular legacy cryptocurrencies such as Bitcoin, which is their main difference

⁵While side-channel attacks are out of the scope of this work, note that FASTKITTEN can achieve forward secrecy of states in case of side-channel attacks using the same mechanism as Ekiden.

compared to our work. Recall that one of the main goals of FASTKITTEN is making minimal assumptions on the underlying blockchain technology and, in particular, to run on top of the Bitcoin blockchain.

Another motivation for off-chain contract execution might be the goal of protecting privacy. Hawk [120] and the Ring of Gyges [104] are examples of works that do keep the state, all inputs and outputs private. It is also true for the scaling solutions mentioned above. These techniques work only over cryptocurrencies with support for complex smart contracts, e.g., over Ethereum. Below we discuss the differences between these solutions and FASTKITTEN when running on top of Ethereum.

TEEs for privacy. None of the solutions discussed above achieves private off-chain contract execution. The work Hawk [120], proposes privacy-preserving contract evaluation, in which the state, all inputs, and all outputs are kept private. Hawk contracts [120] achieve these properties using Ethereum smart contracts that judge computations done by a semi-trusted third party (a manager), who executes the contract on private inputs and is trusted not to reveal any secrets. Initially, all parties submit their encrypted inputs to the contract, then the manager computes the result and proves its correctness with a zero-knowledge proof. If the proof is correct, the contract pays out money accordingly. While the authors of Hawk discuss the possibility of using SGX for instantiating the manager and reducing the trust assumptions in this party, it still leverages the blockchain for every user input, which makes it slower and more costly compared to the execution of FASTKITTEN. We say it only supports single round computation off-chain, which is their main difference to FASTKITTEN. While FASTKITTEN also provides confidentiality of the contract's state and input privacy, our main goal is to enrich cryptocurrencies with no contract support to be able to execute arbitrarily complex smart contracts.

Incentive-driven Verification. An additional direction of related work consists of the Arbitrum [105] and TrueBit [177] projects. Recall from Chapter 3.3 that the Arbitrum protocol lets a set of managers run a smart contract off-chain (in the form of a virtual machine). As long as the managers reach consensus off-chain, the contract state is updated. However, if they disagree, the new state is posted on-chain by one of the managers, and the others can challenge it interactively until the dispute is found. Compared to FASTKITTEN, Arbitrum has different trust assumptions, i.e., that there exists at least one honest manager, requires many more players compared to the single operator of FASTKITTEN and requires

6 Off-Chain Smart Contracts on Bitcoin

significantly more blockchain interaction in case of a dispute.

The TrueBit system [177], as described in Chapter 3.3 works in a similar fashion, only that a single party is randomly assigned to be the solver, who proposes a new off-chain state. The other players act as verifiers and challenge computations of the solver on-chain. Similar to Arbitrum, TrueBit relies on the assumption that there is at least one honest verifier, but it does not keep inputs and the contract state private, even in the optimistic case.

Apart from the different trust models and lower requirements on the underlying blockchain technology, FASTKITTEN differs from Arbitrum and TrueBit by providing stronger privacy guarantees, meaning that in both the optimistic and the pessimistic case, inputs of honest parties, as well as the state of the smart contract, remains private.

Approach	Minimal # TX	Collateral	Generic Contracts	Privacy
Ethereum contracts	$\mathcal{O}(m)$	$\mathcal{O}(n)$	✓	✗
MPC [123, 124, 121]	$\mathcal{O}(1)$	$\mathcal{O}(n^2m)$	✓	✓
TEE based [52, 31, 106]	$\mathcal{O}(m)$	no support for money		✓
FastKitten	$\mathcal{O}(1)$	$\mathcal{O}(n)$	✓	✓

Table 6.1: Selected solutions for contract execution over Bitcoin and their comparison to Ethereum smart contracts. Above, n denotes the number of parties and m is the number of reactive execution rounds.

6.2 Preliminaries

In this chapter, we do not work in the UC model. Instead of using simulation-based definitions, we will analyze this protocol against game-based security definitions. In this section, we provide the formalities and model of the coin mechanics on the ledger, the TEE, and the contract.

Modeling the Contract. Since our framework is not restricted to one specific blockchain, we define a *coin domain* \mathcal{D}_{coin} as a subset of non-negative rational numbers. The concrete definition of the set \mathcal{D}_{coin} depends on the considered blockchain.⁶ In this work, we model an n -party multi-round contract C

⁶For Bitcoin, for instance, it would correspond to the smallest Bitcoin unit, called Satoshi, which is equal to 10^{-8} BTC.

6 Off-Chain Smart Contracts on Bitcoin

as a polynomial-time Turing machine. Every contract has to define a vector $\mathbf{c} \in \mathcal{D}_{dep} = (\mathcal{D}_{coin} \setminus \{0\})^n$ that corresponds to the initial deposits parties are supposed to make. Note that every party must make a deposit, i.e., the coin value has to be positive. We write $C_{\mathbf{c}}$ when a reference to the deposit vector is needed. The contract $C_{\mathbf{c}}$ takes as input a value $\mathbf{state} \in \mathcal{D}_{state} \cup \{\emptyset\}$ and a vector of values $\mathbf{in} \in \mathcal{D}_{in}^n$, and returns an output value $out \in \mathcal{D}_{out}$, a new state $\mathbf{state}' \in \mathcal{D}_{state} \cup \{\perp\}$ and a coin distribution $\mathbf{d} \in \mathcal{D}_{coin}^n$:

$$(out, \mathbf{state}', \mathbf{d}) \leftarrow C_{\mathbf{c}}(\mathbf{state}, \mathbf{in})$$

The initial state of every contract is $\mathbf{state} = \emptyset$ and the special $\mathbf{state} = \perp$ signals the contract's termination. In this final stage, the vector \mathbf{d} defines the final payout to each party of the contract. It must hold that $\sum_{i \in [n]} \mathbf{d}[i] \leq \sum_{i \in [n]} \mathbf{c}[i]$. This restriction guarantees that no money can be created, but we note that the overall account can decrease such that fewer coins are paid out than have been locked. This relates to the case where parties are punished by not giving back their coins. The input domain \mathcal{D}_{in} , the state domain \mathcal{D}_{state} and the output domain \mathcal{D}_{out} are application specific and defined by the contract. For example, in case C is the “Rock-paper-scissor” game, then \mathcal{D}_{in} could be $\{\mathbf{rock}, \mathbf{paper}, \mathbf{scissor}\}$ and the \mathcal{D}_{out} could be $\{\mathbf{winA}, \mathbf{winB}, \mathbf{same}\}$, where the values \mathbf{winA} , \mathbf{winB} would define the winner and \mathbf{same} signals that none of the players won. In this example we only consider a one round game, the $\mathcal{D}_{state} = \emptyset$.

6.2.1 Modeling the Blockchain.

As we want that FASTKITTEN can be deployed over simple cryptocurrencies like Bitcoin, we work in the UTXO model as described in Section 3.1.1. Recall, that we denote a transaction as

$$\mathbf{tx} := (\mathbf{tx.Input}, \mathbf{tx.Output}, \mathbf{tx.Time}, \mathbf{tx.Data}),$$

where $\mathbf{tx.Input}$ refers to the input transaction, $\mathbf{tx.Output}$ denotes the output address, $\mathbf{tx.Value}$ is the coin value, $\mathbf{tx.Time} \in \mathbb{N}$ can specify a timeout and $\mathbf{tx.Data} \in \{0, 1\}^*$ is a data field. Recall also, that a transaction \mathbf{tx} only becomes valid if it is signed with the corresponding secret key of the output address from $\mathbf{tx.Input}$. We require that the underlying blockchain system satisfies three security properties: *liveness*, *consistency* and *immutability* from [84] as defined in Section 3.4.

In order to model interaction with the cryptocurrency, we use a simplified blockchain functionality \mathbf{BC} , which maintains a continuously growing chain of

blocks. It stores a block counter c internally, which starts initially with 0 and is increased on average every t minutes. Every time the counter is increased, a new block will be created, and all parties are notified. To address the uncertainty of the block creation duration, we give the adversary control over the exact time when the counter is increased, but it must not deviate more than $\Delta \in [t-1]$ rounds from t . Whenever any party publishes a valid transaction, it is guaranteed to be included in any of the next $\Delta - k$ blocks. Parties can interact with the blockchain functionality **BC** using the following commands.

- **BC.post(tx)**: If the transaction \mathbf{tx} is valid (i.e., all inputs refer to unspent transactions assigned to creator of \mathbf{tx} and the sum of all output coins is not larger than the sum of all input coins) then \mathbf{tx} is stored in any of the blocks $\{\mathbf{block}_{c+1}, \dots, \mathbf{block}_{c+\Delta-k}\}$.
- **BC.getAll(i)**: If $i < c$, this function returns the latest block count $c - 1$ and a list of blocks that extend b_i : $\mathbf{b} = (b_{i+1}, \dots, b_c)$
- **BC.getLast()**: The function **getLast** can be called by any party of the protocol and returns the last (finished) block and its counter: (c, b_c) .

For every cryptocurrency there must exist a validation algorithm for validating consistency of the blocks and transactions therein, which we model using the function **Extends**. It takes as input, a chain of blocks \mathbf{b} and a checkpoint block b_{cp} and outputs 1 if $\mathbf{b} = (b_{\text{cp}+1}, \dots, b_{\text{cp}+i}, \dots, b_{\text{cp}+i+k})$ is a valid chain of blocks extending b_{cp} and otherwise it outputs 0. In Section 6.6 we give more details on the validation algorithm, and how this function is implemented for the Bitcoin system. Recall, that we assume an adversary which cannot compute a chain of blocks of length k by itself (cf. Section 3.4). This guarantees that he cannot produce a false chain of length i such that this function outputs 1. To make the position of some transaction \mathbf{tx} inside a chain of blocks explicit, we write $\ell := \text{Pos}(\mathbf{b}, \mathbf{tx})$ when the transaction is part of the ℓ -th block of \mathbf{b} . If the transaction is in none of the blocks, the function returns ∞ .

6.2.2 Modeling the TEE.

In order to model the functionality of a TEE, we follow the work of Pass et. al. [159]. We explain here only briefly the simplified version of the TEE functionality, whose formal definition can be found in [159, Fig. 1]. On initialization, the TEE generates a pair of signing keys (mpk, msk) , which we call master public key and

master secret key of the TEE. The TEE functionality has two enclave operations: `install` and `resume`.

- `TEE.install()`: The operation `TEE.install` takes as input a program p which is then stored under an enclave identifier eid . The output of `TEE.install` is the identifier eid and the master public key mpk .
- `TEE.resume(eid, f, in)`: The second enclave operation `TEE.resume` executes the program stored inside an enclave. It takes as input an enclave identifier eid , a function f and the function input in . The output of `TEE.resume` is the result out of the program execution and a quote ϱ over the tuple (eid, p, out) . Since we only consider one instance E of the specific program p , we will simplify the resume command $[out, \varrho] := \text{TEE.resume}(eid, f, in)$ and write:

$$[out, \varrho] := E.f(in)$$

For every attestable TEE there must exist a function `vrfyQuote`(mpk, p, out, ϱ) which on input of a correct quote ϱ outputs 1, if and only if out was outputted by an enclave with master public key mpk and which indeed loaded p . Again, we assume that the adversary cannot forge a quote such that the function `vrfyQuote`() outputs 1. For more information on how this verification of the attestation is done in practice, we refer the reader to [159].

6.3 Security Properties

In this section we present the underlying security goals of FASTKITTEN. We need to consider malicious participants, a malicious operator, or any combination of both. In general the protocol's security must hold even if only a single party is honest. A formal definition and proof can be found in Section 6.5.

For FASTKITTEN we informally state the three security properties *correctness*, *fairness* and *operator balance security*. Intuitively, correctness states that in case all parties behave honestly (including the operator), every party $P_i \in P$ outputs the correct result and earns the amount of coins she is supposed to get according to the correct contract execution. The fairness property guarantees that if at least one party $P_i \in P$ is honest, then (i) either the protocol correctly completes an execution of the contract, (ii) the contract execution does not start and all honest parties get the locked coins back, or (iii) the contract execution is aborted, all honest parties get their invested coins back, and at least one corrupt party gets

punished. Finally, the operator balance security property says that in case the operator behaves honestly, he cannot lose money.

Theorem 3 (Informal statement). *The protocol $\Pi_{\text{FASTKITTEN}}$ as defined in Section 6.4 satisfies correctness, fairness and operator balance security property.*

Proving the fairness property is the most challenging part of the proof. We need to show how honest parties reach consensus on the result of the execution and prove that coins are always distributed between parties according to this result (even if malicious parties collude with the operator). In order to prove the operator balance security, we show that an honest operator has always enough time to publish a valid output transaction which pays him back his deposit, before the time-locked penalty transaction can be posted on the blockchain.

On the security of TEEs. FASTKITTEN’s design depends on a TEE to ensure its confidentiality and integrity. Our design is TEE-agnostic, even if our implementation is based on Intel SGX. Recent research showed that the security and privacy guarantees of SGX can be affected by memory-corruption vulnerabilities [25], architectural [34] and micro-architectural side-channel attacks [180]. We assume that the operator \mathcal{Q} has full control over the machine and consequently can execute arbitrary code with supervisor privileges. While memory corruption vulnerabilities can exist in the enclave code, a malicious operator must exploit such vulnerabilities through the standard interface between the host process and the enclave. For the enclave code, we assume a common code-reuse defense such as control-flow integrity (CFI) [77, 36], or fine-grained code randomization [60, 128] to be in place and active. Architectural side-channel attacks, e.g., based on caches, can expose access patterns [34] from SGX enclaves (and therefore our FASTKITTEN prototype). However, this prompted the community to develop a number of software mitigations [173, 92, 51, 33, 172] and new hardware-based solutions [153, 57, 94]. Microarchitectural side-channel attacks like Foreshadow [180] can extract plaintext data and effectively undermine the attestation process FASTKITTEN relies on, leaking secrets and enabling the enclave to run a different application than agreed on by the parties; however, the vulnerability enabling Foreshadow was already patched by Intel [101]. Since existing defenses already target SGX vulnerabilities and since FASTKITTEN’s design is TEE agnostic (i.e., it can also be implemented using ARM TrustZone or next-generation TEEs), we consider mitigating side-channel leakage as an orthogonal problem and out of scope for this work.

Adversary model. For our protocol we consider a *byzantine adversary* [127], which means that corrupted parties can behave arbitrarily. In particular, this includes aborting the execution, dropping messages, and changing their inputs and outputs even if it means that they will lose money. FASTKITTEN is secure even if n parties are corrupt — in particular this includes the two cases where only the operator is honest, and only one party is honest but the operator is corrupt. We show that no honest party will lose coins, a corrupt party will be penalized and that no adversary can tamper with the result of the contract execution. While we prove security in this very strong adversarial model, we additionally observe that incentive-driven parties (i.e., parties that aim at maximizing their financial profits) will behave honestly, which significantly boosts efficiency of our scheme.

6.4 The FastKitten Protocol

In this section, we give a more detailed description of our protocol, which includes the specification of the protocol run by \mathcal{Q} and honest parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, all transactions, and a description of the enclave program FASTKITTEN. The interaction between $\mathcal{Q}, \mathcal{P}_i$, and the blockchain is depicted in Figure 6.2.

The FASTKITTEN protocol proceeds in three phases. During the *setup phase*, the contract is installed in the enclave, attested, and all parties deposit their coins. Then the *round execution* follows for all m rounds of the interactive contract. When the contract execution aborts or finishes, the protocol enters the *finalize phase*. We now explain all phases and the detailed protocol steps for all involved parties and the operator \mathcal{Q} in depth. The detailed interactions as well as the subprocedure of the parties and the operator are displayed in Figure 6.2 and the FASTKITTEN enclave program p_{FK} is displayed below. Overall the protocol requires six different types of transactions.

6.4.1 Setup Phase

In the setup phase, each party \mathcal{P}_i first runs the **Initialize** subprocedure to generate its key pairs and gets the latest block b_{cp} , which serves as a genesis block or checkpoint of the protocol. Then \mathcal{P}_i sends the set of parties P , the b_{cp} , and the contract C to the operator \mathcal{Q} . Upon receiving the initial values from all n parties, \mathcal{Q} runs the subprocedure **InitEnclave** to initialize the trusted execution of the enclave program $p_{\text{FK}}(P, C, \kappa, b_{\text{cp}})$ where κ is the security parameter of the scheme. This security parameter κ also determines the values for the timeout period of t , and

6 Off-Chain Smart Contracts on Bitcoin

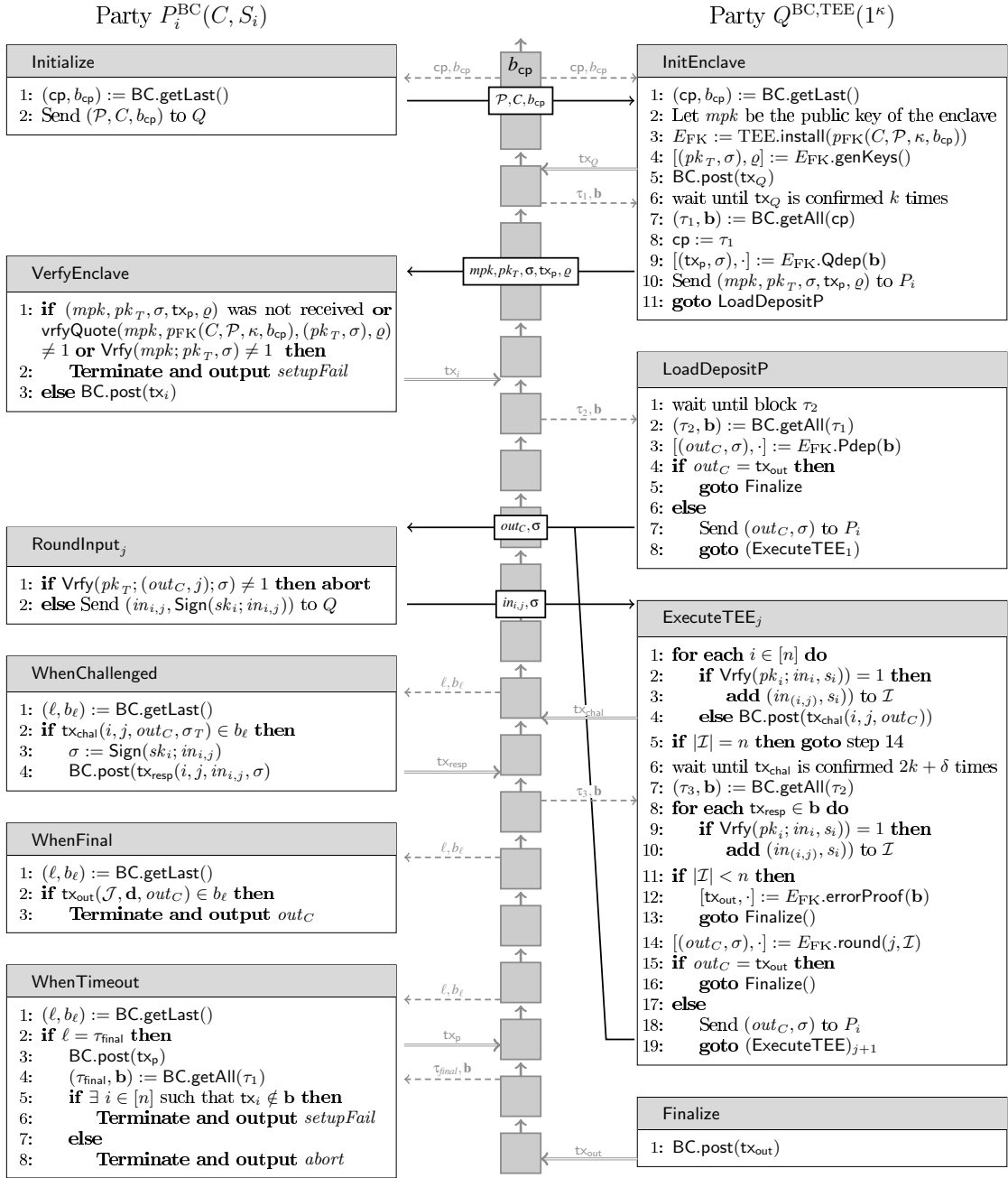


Figure 6.2: Protocol FASTKITTEN. Direct black arrows indicate communication between the parties and Q , gray dashed arrows indicate reading from the blockchain and gray double arrows posting on the blockchain.

6 Off-Chain Smart Contracts on Bitcoin

FastKitten enclave program $p_{\text{FK}}(P, C, \kappa, b_{\text{cp}})$
<p>The execution of p_{FK} is initialized with the secret key msk, the set of parties (where every $\mathcal{P}_i \in P$ is identified by its key pk_i), a contract C, a security parameter κ (which also defines the waiting period t and confirm period k) and a checkpoint b_{cp}. Internally it stores the contract state and the status flag s initially set to $\text{state} = \emptyset$ and $s = s_{\text{genKeys}}$.</p>
function $\text{genKeys}()$
<pre> 1: if $s \neq s_{\text{genKeys}}$ then abort 2: store $(sk_T, pk_T) := \text{Gen}(1^\kappa)$ 3: set $s := s_{\text{Qdep}}$ 4: return $pk_T, \text{Sign}(msk; pk_T)$ </pre>
function $\text{Qdep}(\mathbf{b})$
<pre> 1: if $s \neq s_{\text{Qdep}}$ or $\text{Extends}(b_{\text{cp}}, \mathbf{b}) \neq 1$ or $\text{Pos}(\mathbf{b}, \text{tx}_Q) > \mathbf{b} - k$ then abort 2: set $s := s_{\text{Pdep}}$ 3: set $b_{\text{cp}} :=$ last block of \mathbf{b} 4: return tx_p </pre>
function $\text{Pdep}(\mathbf{b})$
<pre> 1: if $s \neq s_{\text{Pdep}}$ or $\text{Extends}(b_{\text{cp}}, \mathbf{b}) \neq 1$ then abort 2: set $\mathbf{J} := \emptyset$ 3: for $i \in P$ do 4: $\ell_i := \text{Pos}(\mathbf{b}, \text{tx}_i)$ 5: if $\ell_i < \delta$ and $\ell_i < \mathbf{b} - k$ then add i to \mathbf{J} 6: if $\mathbf{J} = n$ then 7: $s := s_{\text{round1}}$ 8: $b_{\text{cp}} := \mathbf{b}.\text{last}$ 9: return $\emptyset, \text{Sign}(sk_T; \emptyset, b_{\text{cp}})$ 10: else 11: $s := s_{\text{terminated}}$ 12: return $\text{tx}_{\text{out}}(\mathbf{J}, \mathbf{c}, \text{setupFail})$ </pre>
function $\text{round}(j, (in_1, \sigma_1) \dots, (in_n, \sigma_n))$
<pre> 1: if $s \neq s_{\text{round}j}$ or for any $i \in [n] : \text{Vrfy}(pk_i; in_i, \sigma_i) \neq 1$ then abort 2: $(out_C, \text{state}', \mathbf{d}) := C(\text{state}, \vec{in})$ 3: if $\text{state}' \neq \perp$ then 4: $s := s_{\text{round}j+1}$ 5: $\text{state} := \text{state}'$ 6: return $(out_C, \text{Sign}(sk_T; (out_C, j)))$ 7: else 8: $s := s_{\text{terminated}}$ 9: return $\text{tx}_{\text{out}}([n], \mathbf{d}, out_C)$ </pre>
function $\text{errorProof}(j, \mathbf{b})$
<pre> 1: if $s \neq s_{\text{round}j}$ or $\text{Extends}(b_{\text{cp}}, \mathbf{b}) \neq 1$ then abort 2: Let $\sigma := \text{Sign}(sk_T; (out_C, j))$ 3: $\mathbf{J} := [n]$ 4: for $i \in P$ do 5: if $\text{Pos}(\mathbf{b}, \text{tx}_{\text{chal}}(i, j, out_C, \sigma)) < \mathbf{b} - \delta - k$ then 6: if $\text{Pos}(\mathbf{b}, \text{tx}_{\text{resp}}(i, j, in, \sigma)) > \mathbf{b} - k$ then 7: delete i from \mathbf{J} 8: else if $\text{Vrfy}(pk_i; in, \sigma) \neq 1$ then 9: delete i from \mathbf{J} 10: $s = s_{\text{terminated}}$ 11: if $\mathbf{J} \neq n$ then 12: return $\text{tx}_{\text{out}}(\mathbf{J}, \mathbf{c}, \text{abort})$ </pre>

6 Off-Chain Smart Contracts on Bitcoin

the confirmation constant k . This ensures that all parties and the TEE agree on these fixed values. Once p_{FK} is installed in the enclave, it generates key pairs for the protocol execution and, in particular, the blockchain public key pk_T ⁷. Now, \mathcal{Q} can make its deposit transaction $\text{tx}_{\mathcal{Q}}$ which assigns q coins to the enclave public key (cf. Figure 6.3a). Let block counter τ_1 denote the time when this transaction

\mathcal{Q}'s Deposit Transaction $\text{tx}_{\mathcal{Q}}$	\mathcal{P}_i's Deposit Transaction tx_i
tx.Input: Some unspent tx from \mathcal{Q}	tx.Input: Some unspent tx from \mathcal{P}_i
tx.Output: Assign q coins to TEE	tx.Output: Assign c_i coins to TEE

(a) \mathcal{Q} 's deposit of q coins (b) \mathcal{P}_i 's deposit of c_i coins

Figure 6.3: Deposit transactions issued by the operator \mathcal{Q} during the Setup phase.

has been included and confirmed in the blockchain. \mathcal{Q} loads all blocks from cp to τ_1 as evidence to the enclave. If this evidence is correct, the execution of p_{FK} function Qdep outputs a penalty transaction tx_{p} as depicted in Figure 6.4. This transaction will only be valid after timeout τ_{final} (after which the protocol must be terminated) has passed and pays out the q coins of \mathcal{Q} 's deposit transaction $\text{tx}_{\mathcal{Q}}$ to the parties $\mathcal{P}_1, \dots, \mathcal{P}_n$. This transaction is used whenever the protocol does not finish before the final timeout τ_{final} , which equals $(3 + 2m) \times \Delta$ blocks after the protocol start (recall the time model of Section 3.4).⁸ \mathcal{Q} sends the penalty

Penalty Transaction tx_{p}	
tx.Input:	\mathcal{Q} 's Deposit Transaction $\text{tx}_{\mathcal{Q}}$
	For all $i \in [n]$:
tx.Output _{i} :	Assign c_i coins to \mathcal{P}_i
tx.Time:	Spendable after τ_{final}

Figure 6.4: Penalty transactions issued by the enclave during the setup.

transaction to all parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, who run subprocedure VerifyEnclave . A party will only proceed if it received this penalty transaction from \mathcal{Q} during the setup and verified that the program $p_{\text{FK}}(P, C, \kappa, b_0)$ is installed in the enclave (through the attestation process). Only if all checks pass, it creates and publishes its deposit transaction tx_i (cf. Figure 6.3b). After time $\tau_2 < \tau_1$, \mathcal{Q} executes LoadDepositP and

⁷For simplicity we omit here, that the enclave might use different key pairs for signing transactions and messages

⁸The definition of τ_{final} guarantees that even if the execution is delayed in every round, an honest operator will not be penalized.

again provides the block evidence to the enclave execution of p_{FK} . If all parties published the deposit transactions, the first-round execution starts. Otherwise, the enclave proceeds to the finalize phase and outputs a refund transaction $\text{tx}_{\text{out}}(T, \vec{c})$ that returns the deposit back to honest users and \mathcal{Q} , where $T \subset \mathcal{P}$ is the set of all parties that submitted the deposit transaction until time τ_2 . Note that the internal state of the contract execution is maintained by the p_{FK} program inside the enclave. This guarantees that the contract is not executed on an outdated state.

6.4.2 Round Computation Phase

When the protocol arrives in the round computation phase, \mathcal{Q} sends the authenticated output of the enclave to every party \mathcal{P}_i and requests input for the next round. Each party \mathcal{P}_i runs the `round` algorithm. Internally it verifies whether the input request came from the enclave by verifying the attached signature. Then it generates and signs its round input and sends it to \mathcal{Q} . While \mathcal{P}_i waits for the next round, \mathcal{Q} verifies all received inputs and their signatures in the `ExecuteTEE` subprocedure. All inputs of the parties \mathcal{P}_i that responded with correctly signed round inputs, are sorted in the round input set I . Once all parties sent their inputs, \mathcal{Q} triggers the execution of the contract in the enclave. Let us emphasize that in this simplified description of our protocol, we do not focus on the privacy aspect. Hence, we omit that all round inputs to the contract could be encrypted with the public key of the enclave. In this case, the trusted enclave execution needs to decrypt them before it evaluates the contract on them. See Section 6.7.1 for more details.

Note that the operator \mathcal{Q} may be malicious and refrain from requesting a party \mathcal{P}_i for the input to a round computation. Instead, \mathcal{Q} may pretend that it actually did not receive any input from the party \mathcal{P}_i . On the other hand, one can imagine a scenario where \mathcal{Q} is behaving honestly, but the party \mathcal{P}_i is dishonest and does not send the correctly signed round input to \mathcal{Q} . Note that the program p_{FK} cannot distinguish between these two cases without additional information. We will next show how an honest \mathcal{Q} can generate a proof to attribute the malicious behavior to \mathcal{P}_i . If it is not able to generate this proof within some time, the enclave will penalize \mathcal{Q} instead. First, \mathcal{Q} has to publish a challenge transaction tx_{chal} , which includes the signed output of the previous step. tx_{chal} spends a tiny amount μ of coins from \mathcal{Q} and assign them to party \mathcal{P}_i ⁹.

⁹Cryptocurrencies like Bitcoin allow transactions with very small denominations (e.g., fractions

6 Off-Chain Smart Contracts on Bitcoin

Transaction $\text{tx}_{\text{chal}}(i, j, \text{out}_C, \sigma_T)$	Transaction $\text{tx}_{\text{resp}}(\mathbf{i}, \mathbf{j}, \text{in}, \sigma_{\mathbf{i}})$
tx.Data: Store $i, j, \text{out}_C, \sigma_T$	tx.Data: Store $i, j, \text{in}, \sigma_{\mathbf{i}}$
tx.Input: Some unspent tx from \mathcal{Q}	tx.Input: $\text{tx}_{\text{chal}}(i, j, \text{state})$
tx.Output: Assign μ coins to \mathcal{P}_i	tx.Output: Assign μ coins to \mathcal{Q}

(a) \mathcal{Q} 's challenge transaction
(b) \mathcal{P}_i 's response transaction

Figure 6.5: Challenge response transactions for the i -th round.

Once tx_{chal} is included in the blockchain, party \mathcal{P}_i can read the correct output information from the transaction. The party should respond with tx_{resp} , which includes its signed round input. tx_{resp} spends the tx_{chal} and assigns the μ coins back to \mathcal{Q} . The action of \mathcal{P}_i is depicted via the `WhenChallenged` subprocedure. If the party \mathcal{P}_i fails to post tx_{resp} on the blockchain, \mathcal{Q} can feed proof about this into the TEE and the computation aborts while \mathcal{P}_i is monetarily penalized. If instead \mathcal{P}_i responds, \mathcal{Q} will take the correct input information from the transaction and give it to the TEE which will execute the next round of the contract. If some party does not send the response after it was challenged within δ blocks, \mathcal{Q} can prove this misbehavior to the FASTKITTEN program, by providing the blockchain evidence of the challenge-response transcript. If the enclave program identifies a cheating party via this evidence, it proceeds to the finalize phase. Otherwise, if all the parties' inputs were received with authentication (possibly after the challenge-response phase), \mathcal{Q} instructs the enclave to execute the contract on the accumulated input. The result of the contract execution is the output out_C , the updated state state , and a coin distribution denoted by \mathbf{d} . If state equals \perp , the contract execution is finished, and the protocol proceeds to the finalize phase. Otherwise, FASTKITTEN internally stores the state and outputs out_C to \mathcal{Q} , who sends this output to all parties and waits for next round inputs.

6.4.3 Finalize Phase

In the finalize phase, the enclave publishes a final output transaction tx_{out} which distributes the coins back to all honest parties (cf. Figure 6.6). It is parameterized by a set of parties to receive coins \mathbf{J} , a final coin distribution \vec{e} and a final state out_C . The transaction $\text{tx}_{\text{out}}(\mathbf{J}, \vec{e}, \text{out}_C)$, spends all deposit transactions tx_i for all $i \in \mathbf{J}$ and \mathcal{Q} 's deposit transaction $\text{tx}_{\mathcal{Q}}$. It includes the out_C in the data field and assigns q coins back to \mathcal{Q} and e_i coins to party \mathcal{P}_i , for every $i \in \mathbf{J}$. Let us

of cents).

6 Off-Chain Smart Contracts on Bitcoin

Output Transaction $\mathbf{tx}_{\text{out}}(\mathbf{J}, \vec{e}, \text{out}_C)$	
tx.Data:	Store out_C
tx.Input:	Deposit Transactions $\mathbf{tx}_Q, \{\mathbf{tx}_i\}_{i \in \mathbf{J}}$
tx.Output₁:	q coins to Q For all $i \in \mathbf{J}$:
tx.Output_{$i+1$}:	e_i coins to \mathcal{P}_i

Figure 6.6: Output transaction issued by the FASTKITTEN enclave.

note that $\mathbf{J} = [n]$ implies correct protocol termination. If $\mathbf{J} \neq [n]$, then some party misbehaved and the protocol failed. Either a party did not make a deposit in the setup phase (signaled by $\text{out}_C = \text{setupFail}$) or some party aborted in the round computation phase (signaled by $\text{out}_C = \text{abort}$). In both cases, all other parties get their initial deposits back. Note, that if a party \mathcal{P}_j is caught cheating by the TEE, it will lose its deposit. Q now has to publish this transaction to get his coins before time τ_{final} and by that also distributes coins and reveals out_C to honest parties. The participants need to continually monitor the blockchain for transactions that challenge them or indicate the final output. When they see a challenge transaction, they respond as described above. If they see an output transaction, they know the protocol execution ended and output the final contract output according to subroutine `WhenFinal`.

6.5 Security Evaluation

In this section, we informally argue how FASTKITTEN fulfills the security properties proposed in Section 6.3. For a full proof, we refer the reader to [59]. Here we just outline the main ideas and challenges of the proof. The formal security statement that we prove in [59] is as follows:

Theorem 3 (Formal statement). *Let $(\text{Gen}, \text{Sign}, \text{Vrfy})$ be a signature scheme that is existentially unforgeable under chosen message attack, a trusted execution environment emulating the TEE ideal functionality (as modeled in Sec. 6.2.2) and a blockchain emulating the BC ideal functionality (as modeled in Sec. 6.2.1), the protocol $\pi_{\text{FASTKITTEN}}$ as defined in Section 6.4 satisfies the properties correctness, fairness, and operator balance security property.*

Let us now take a closer look at these properties.

Correctness. The FAIRSWAP protocol is correct if in case where all parties are honest and follow the protocol description, the contract will execute correctly with output out and the coin distribution out will be enforced. In order to formally prove this property, we first need to define what it means to have evaluated the contract correctly. For this reason, we specify an evaluation algorithm that applies the inputs of all contract participants on the state of a contract to receive a new state [59]. Then we can show correctness through the following steps.

1. All honest parties will agree on the same contract C during setup. If the operator honestly follows the protocol, all parties will lock their coins during the setup, and the protocol proceeds to the round computation phase.
2. All parties will send their inputs, and the operator will evaluate the contract C inside the enclave. If all parties follow the protocol, the new state of the contract after every round will always be equal to the one that the ideal evaluation algorithm would compute.
3. An honest operator will enforce the final output $(\text{out}_C, \perp, \mathbf{d})$ by publishing the output transaction $\text{tx}_{\text{out}}(\mathcal{P}, \mathbf{d}, \text{out}_C)$. This transaction will rightfully distribute d to each party.

Together these steps ensure correctness, i.e., that a contract C is executed correctly and enforced on-chain if all parties, including the operator are honest.

Operator balance security. This property ensures that no operator who behaves honestly and follows the protocol description will lose money. In order to show operator balance security, we show that an honest \mathcal{Q} can always get his coins back before the penalty transaction becomes valid and can appear on the blockchain. This means the TEE will always output an output transaction in time which pays q coins back to \mathcal{Q} .

If all parties are honest, the output transaction will ensure that the operator gets his funds back (cf. correctness). But the more interesting case is if the operator is honest, but the parties are not. If at least one of them does not cooperate in the *setup phase*, e.g., because he does not agree on the contract C , the enclave aborts the contract execution. Because an honest operator will have locked the required balance, the enclave outputs a refund transaction, that allows \mathcal{Q} to claim his entire deposit back. This will happen at the end of the setup phase, so the refund transaction will never be valid. Next, we consider the scenario where potentially malicious parties act honestly during the setup phase but start misbehaving in

the round computation phase. For every party that does not follow the protocol execution, the operator might need to challenge the parties' input. If during any round, a party does not send the required inputs, the operator can prove this fact to the enclave. In this case, the contract execution stops, and the operator can always enforce the refund transaction before the penalty transaction gets valid. Therefore, in all cases, an honest operator will receive his locked coins back.

Fairness. The fairness property guarantees that if at least one party is honest, one of the following three cases is guaranteed to happen:

Case 1: Either the protocol correctly completes the contract execution, and the contract output is enforced. The operator will get his deposit back.

Case 2: All honest parties (including the operator) output *setupFail* and get their invested coins back.

Case 3: All honest parties output **abort** and get their invested coins back, and at least one misbehaving party gets punished.

Case 1 occurs in the honest case, and the correctness property guarantees that transaction \mathbf{tx}_{out} is published by \mathcal{Q} . We additionally show that an honest party outputs $\mathbf{out} \notin \{\mathit{setupFail}, \mathbf{abort}\}$ only if a transaction $\mathbf{tx}_{\text{out}}(\mathbf{J}, \vec{e}, \mathit{out})$ was published on the blockchain. Since this transaction spends the deposit transactions, which can only be spent once, all honest parties will reach a consensus on the output value *out*. We conclude the proof for case 1 by showing that *out* and the final coin distribution \vec{e} has to be equal to the correct result of the contract execution.

Our proof strategy for cases 2 and 3 of the fairness property is to first prove that if at least one honest party outputs *setupFail* or **abort**, then all parties output *setupFail* or **abort** respectively. Case 2 is enforced by the fact that if one party outputs *setupFail*, either the operator sends the refund transaction by the end of the setup phase, or the penalty transaction is invoked. As both transactions refund any possible deposits, all honest parties that sent a deposit will stay financially neutral and also output *setupFail*. If any party never deposited coins, its financial neutrality is ensured, and it will terminate with the result *setupFail*.

The proof for case 3 is identical, except for that we consider the parties' output **abort** and the possible public transactions are either the penalty transaction or the error proof transaction $\mathbf{tx}_{\text{out}}(\mathbf{J}, \mathbf{c}, \mathbf{abort})$. Additionally, we need to show that at least one malicious party that does not get its funds back. This is either the operator, who loses his deposit through the penalty transaction or one malicious

party which did not send inputs. By definition of the function `errorProof(j, b)` and the unforgeability of the underlying signature scheme, the enclave will only output this transaction if misbehavior happened and also ensure that the responsible party is punished.

6.6 Implementation and Performance

We also implemented the FASTKITTEN protocol for the Bitcoin blockchain over an Intel SGX enclave. We refer the reader to [59] for a full specification of the implementation details and design choices and only provide a high-level overview of the main challenges and its performance here.

6.6.1 Implementation Challenges

Running code inside a TEE enclave is less efficient than the straightforward execution on a “normal” device. Therefore we try to keep the code of the FASTKITTEN enclave as efficient as possible. Here we discuss how we process the interaction between the TEE and the blockchain and how we deal with contracts that do not halt.

Blockchain Verification

One challenge for the implementation of the FASTKITTEN enclave inside the TEE is a correct but efficient block verification. As running the TEE as a Bitcoin full node is neither efficient nor necessary for the protocol, we needed to specify the most minimalistic requirements for secure verification.

One measure to simplify the execution is defining a custom genesis block for the protocol execution. If we let all parties reach consensus on this block, and at least one party is honest, this block is a valid and confirmed block of the chain but also not too far in the past to make verification of follow up blocks much faster. If a malicious operator proposed an insecure genesis block, an honest party would not accept it, and the protocol would not start. Additionally, the verification of blocks can be sped up by letting the TEE store the latest accepted block-hashes such that at any later point, it only has to verify the correctness of new blocks.

Another measure to simplify the block validation is the fact that often, the TEE does not need to verify all transactions inside a block but only some information. In the setup and the finalization phase, it needs to check the integrity of blocks

it gets from the operator and verify if a specific expected transaction is inside it. Thus, it is sufficient to verify that these transactions are part of a valid block – without downloading entire blocks, which can be done efficiently using simplified payment verification (SPV).

However, SPV libraries can only prove that a transaction *is* part of a block on the blockchain, but they cannot prove that a transaction *is not* part of any block. As required by the challenge-response case, we added an alternative verification mode that fully downloads every block that could potentially contain the transaction and checks whether it appears in any of those blocks. This verification algorithm is slower and less efficient, but it is only required during the response verification, which happens only in the pessimistic case and at most once. In measurements, we estimated the time for the verification of a single block to be around 5 seconds inside an Intel SGX [59].

Denial of Service Protection

The FASTKITTEN protocol assumes immediate contract execution meaning that the execution of a contract inside a TEE takes no time. For most practical contracts, this simplifying assumption is reasonable since executing a simple contract function inside a TEE is much faster than waiting for it to be evaluated on-chain. However, this is not true when considering arbitrary contracts which might potentially contain endless loops. Moreover, the halting problem states that it is impossible to predict if a certain algorithm will halt within a certain number of steps. A simple protection against endless loops and denial-of-service attacks is letting the enclave monitor the execution of the smart contract and terminate execution if the number of execution steps exceeds a predefined limit. If the contract execution is aborted due to an execution timeout, the enclave signs an output transaction tx_{out} , which returns deposited coins back to parties and to the operator.

6.6.2 Performance

Let us now take a look at FASTKITTEN’s performance. We describe the number of transactions, the deposits, and the number of rounds for every phase. We assume that it takes one round to send direct messages and up to Δ rounds for interactions with the blockchain (cf. Section 3.4). Note that it is only an upper bound, and the rounds can and will most likely take less time. We also analyze the fees for every step, where the overview of the transaction fees for the FASTKITTEN transactions

6 Off-Chain Smart Contracts on Bitcoin

can be found in Table 6.2. Recall, that we consider an exchange rate of 7951.95 and a transaction fee of 15 satoshi per byte (cf. Section 3.1.3).

Transaction	Size [Bytes]	Fees		
		[satoshi]	[BTC]	[EUR]
Deposit ($\text{tx}_{\mathcal{Q}}, \text{tx}_i$)	250	3750	0.0000375	0.30
Penalty (tx_p)	504	7560	0.0000756	0.60
Challenge (tx_{chal})	293	4395	0.00004395	0.35
Response (tx_{resp})	266	3990	0.0000399	0.32
Output (tx_{out})	1986	29790	0.0002979	2.37

Table 6.2: Estimated fees for a typical deposit transaction and the FASTKITTEN transactions.

During the setup phase, each party \mathcal{P}_i deposits c_i coins, and the operator needs to deposit an amount $\sum_{i \in [n]} c_i$ which equals the sum of all other deposits from P together. To post the deposit transactions $\text{tx}_1, \dots, \text{tx}_n$ and $\text{tx}_{\mathcal{Q}}$, a total of $n + 1$ transactions are necessary. However, as all parties can send their deposits simultaneously, the setup phase takes $2\Delta + 2$ rounds in the optimistic case, additionally to the time necessary for the attestation process (from which we abstract in the protocol description). During the setup phase, every party approximately pays 0.30 euros in fees.

During the round computation phase, in the optimistic case, FASTKITTEN can operate completely off-chain without any blockchain interaction. In the best case, this phase takes $2m$ rounds, one for sending the input and one for the output for an m -round contract. In the pessimistic case, any user might withhold their input in any given round or the operator challenges parties. If this (pessimistic) case occurs, the challenge-response procedure requires transactions also in the round computation phase. In the worst-case scenario, this happens in every round for every party and which can lead to $2nm$ additional transactions and execution time of up to $2m\Delta$ rounds. We note however, that as no party would benefit from this scenario it is a highly unlikely case, when rational parties are considered. The value of coins which are sent in every transaction is very low, so the main overhead of this procedure are the transaction fees (approx. 35 euro cent per transaction) and delays.

In the finalize phase, FASTKITTEN requires one additional payout transaction tx_{out} to settle money distribution among the parties. The transaction fee is paid by the operator, and its size can vary depending on the use case. For a poker game,

The output is roughly 1986 Bytes, which results in fees over 2 euros. The operator will most likely add these costs to his fees (we discuss this in more detail in the next section). In case the operator misbehaves and does not/ cannot finish the contract evaluation inside the enclave, the penalty transaction \mathbf{tx}_p becomes valid after at most $\tau_{\text{final}} = (3 + 2m)\Delta$ rounds. This means the protocol will take in the worst case $2 + (5 + 2m)\Delta$ rounds. The fees for the penalty transaction (0.6 euros) need to be paid by one of the users. When no measure to share the costs is in place, this fee will be carried by one of the honest parties, as all of them would submit the transaction but only one of them will be included in the blockchain.

6.7 Discussion and Extensions

In order to explain and analyze the FASTKITTEN protocol, we presented the basic protocol version, which includes the essential building blocks required to guarantee security. Depending on the use case, one might be interested in further properties. Possible extensions discussed in this section include the option to pay the operator for his service, protect the operator against TEE faults, hide the contract output from \mathcal{Q} or eavesdroppers and allow cross-currency smart contracts.

6.7.1 Privacy

As mentioned in the introduction, traditional smart contracts cannot preserve the privacy of user inputs and thus always leak internal data to the public. In contrast to common smart contract technologies, the FASTKITTEN protocol supports privacy-preserving smart contracts as proposed in Hawk [120]. This requires *private contract state* to hide the internal execution of the contract and *input privacy*, which means that no party (including the operator) sees any other parties' round input before sending its own. It is straightforward to see that FASTKITTEN has a secret state since it is stored and maintained inside the enclave. Input privacy can easily be achieved by encrypting all inputs with the public key of the enclave. This guarantees that only the FASTKITTEN execution facility and the party itself knows the inputs. If required, FASTKITTEN could also be extended to support privacy of outputs from the contract to the parties by letting the enclave encrypt the individual outputs with the parties' public keys. But this additional layer should only be used when the contract requires it since, in the worst case, this increases the output complexity of the challenge and output transaction.

6.7.2 Applications

FASTKITTEN allows running complex smart contracts on top of cryptocurrencies that do not natively support such contracts, like Bitcoin. However, in contrast to Turing-complete contract execution platforms like Ethereum, a secure off-chain execution such as FASTKITTEN puts some restrictions on the contracts it can run:

- The number of parties interacting with the contract must be known at the start of the protocol.
- It must be possible to estimate an upper bound on the number of rounds and the maximum run time of any round.

All of these restrictions make FASTKITTEN contracts different from smart contracts running on Ethereum itself. Other off-chain solutions (like state channels (cf Section 3.3) come with similar caveats. By allowing additional blockchain interactions, we could get around those restrictions, but we would lose efficiency in the optimistic case (which is also similar to state channel constructions). FASTKITTEN has important features that are supported by neither Bitcoin nor Ethereum — FASTKITTEN allows private inputs and batched execution of user inputs. Overall, this leads to cheaper, faster, and private contract execution than what is possible with on-chain contracts in Ethereum. Below, we highlight these efficiency gains by presenting four concrete use-cases in which FASTKITTEN outperforms contracts run over Ethereum or in Ethereum state channels.

Lottery. A lottery contract takes coins from every involved party as input, and randomly selects one winner, who gets all the coins. The key challenge for such a contract is to generate randomness to select the winner in a fair way. In Ethereum or Bitcoin the randomness is computed from user inputs through an expensive commit-reveal scheme [147]. In FASTKITTEN, all parties can immediately send their random inputs to the enclave, which will securely determine a winner. Hence, we reduce the round complexity from $\mathcal{O}(\log n)$ [147] to $\mathcal{O}(1)$.

Auctions. Another interesting use-case for smart contracts are auctions, where parties place bids on how much they are willing to pay, and the contract determines the final price. In a straightforward auction, the bids can be public, but more fair versions, like second bid auctions, require the users not to learn the other bids before they place their own. The privacy features of FASTKITTEN can be used

to reduce the round complexity for such auctions, which would otherwise require complex cryptographic protocols [83].

Rock-paper-scissors. We implemented the popular two-party game rock-paper-scissors to show the feasibility of FASTKITTEN contracts. Again, the privacy features allow one match to be executed in a single round, which would have required at least three rounds in Ethereum. The pure execution time in the optimistic case, excluding delays due to human reaction times, is 12ms for one round (averaged over 100 matches). This demonstrates that off-chain protocols, like FASTKITTEN, are highly efficient when the same set of parties wants to run complex contracts (like multiple matches of a game).

Poker. We also implemented a Texas Hold'em Poker game, to prove that multi-party contracts which inherently require multiple rounds can also be efficiently executed in FASTKITTEN. In our implementation, each player starts with an equal deposit and participates in (at least one) initial betting round. The cards are shuffled and distributed privately by the enclave. As the game proceeds, the enclave determines and reveals the winner, and correctly distributes the chips of the current pot. The game continues until only one player remains. We measured 50 matches between 10 players resulting in an average time of 45ms per match (multiple betting rounds are included in each match). The run time was measured starting from the moment all deposits are committed to the blockchain (details on the exact measurements and analysis can be found in [59]).

6.7.3 Fees for the Operator

The owner of the TEE provides a service to the users who want to run a smart contract, and, naturally, he wants to be paid for it. In addition to the costs of buying, maintaining, and running the trusted hardware, he also needs to block the security deposit q for the duration of the protocol. While the security of FASTKITTEN ensures that he will never lose this money, he still cannot use it for other purposes. The goal of the operator-fees is to make both investments attractive for \mathcal{Q} . We assume that the operator will be paid ξ coins for each protocol round for each party. Since the maximum number of rounds m is fixed at the protocol start, \mathcal{Q} will receive $\xi \times n \times m$ coins if the protocol succeeds (even if the contract terminated in less than m rounds). If the operator proves to the TEE in round x that another party did not respond to the round challenge, he

will only receive a fee for the passed x number of rounds (namely $\xi \times x \times n$). This pay-per-round model ensures that the operator does not have any incentive to end the protocol too early. If the protocol setup does not succeed or the operator cheats, he will not receive any coins. The extended protocol with operator fees requires each party to lock $c_i + m \times \xi$ coins and the operator needs to level this investment with $qc_i + m \times \xi$ coins.

6.7.4 Incentive-driven Adversary

While this work does not include game-theoretic analysis of the designed protocols, we can still highlight the effects that the protocol design can have on incentive-driven adversaries. The ideal outcome of a formal analysis is that the expensive pessimistic protocol case will never occur. Hence, if the setup phase completes successfully, then the result of the protocol is a correct contract execution. To understand the incentive mechanics that are in place to prevent the pessimistic case let us take a closer look on the implications for misbehaving parties. When the operator cheats, he is punished by losing all his security deposits. This is a strong incentive for him to follow the protocol. If the protocol aborts due to a missing input of a misbehaving party, this party will lose its coins. This again, is by definition of incentive-driven parties, against their interest. Even when multiple parties cheat, there is still a chance that any one of them will be punished. Even preventing the challenge-response procedure is incentivized when we consider the fees for posting transactions on the blockchain. These additional incentives enforce fast and protocol compliant behavior of the parties, when their only goal is to maximize their financial gain. However, if the main goal of an attacker is to hurt honest parties or enforce a certain smart contract outcome, his incentives are less easy to analyze.

6.7.5 Fault Tolerance

In order to ensure that the execution of the smart contract can proceed even in the presence of software or hardware faults, the enclave can save a snapshot of the current state in an encrypted format, e.g., after every round of inputs. This encrypted state would be sent to the operator and stored on redundant storage. If the enclave fails, the operator can instantiate a new enclave, which will restart the computation, starting from the encrypted snapshot. If the TEE uses SGX, snapshots will leverage SGX's sealing functionality [100] to protect the data from

the operator while making it available to future enclave instances.

6.7.6 Multi-Currency Contracts

FASTKITTEN requires from the underlying blockchain technology that transactions can contain additional data and can be time-locked. Any blockchain like Bitcoin, Ethereum, Lightcoin, and many others that allow these transaction types can be used for the FASTKITTEN protocol. With some minor modifications, FASTKITTEN can even support contracts that can be funded via multiple different currencies. This allows parties that own coins in different currencies to still execute a contract (play a game) together. The main modification to the FASTKITTEN protocol is that the operator and the enclave need to simultaneously handle multiple blockchains. In particular, for each of the considered currencies, \mathcal{Q} needs to deposit the sum of all coins that were deposited by parties in that currency. This is in order to guarantee that if the operator cheats, players get back their invested coins in the correct currency. In addition, the operator is obliged to challenge each party via its blockchain. If the execution completes (or the operator proves to the enclave that one of the players cheated), the enclave signs one output transaction for each of the currencies. While this extension adds complexity to the enclave program and leads to more transactions and thus transaction-fees, the overall deposit amount stays identical to the single blockchain use case.¹⁰ The complete design and proof of correctness of a cross-ledger FASTKITTEN is left to future work.

¹⁰This solution assumes that any party can receive coins in any of the considered currencies.

7 Conclusion

In this thesis, we discussed three optimistic protocols for increasing blockchain scalability. All three protocols take the off-chain approach, in which transactions and contract evaluations are processed locally on the devices of the protocol participants instead of globally on the blockchain. The motivation behind all three protocols is to increase execution speed and reduce the costs of different blockchain applications.

All proposed protocols, PERUN, FAIRSWAP, and FASTKITTEN, guarantee that no coins can be stolen from honest parties. To achieve this, each party has to lock the required amount of coins on the blockchain at the protocol start and, in particular, before sending off-chain transactions. At the end of the protocol execution, the funds are unlocked and redistributed according to the rules of the protocol. All protocols have a fixed runtime (measured in rounds) to ensure that honest parties reliably know the point in time when they are able to claim back their locked coins.

To guarantee that no coins can get stolen, we analyzed and proved the security of the protocols. At the same time, we aim to build cost- and time-efficient protocols. For this purpose, we consider different cases of (mis-)behaving parties. In particular, the optimistic case that occurs if all parties behave honestly and the pessimistic case¹ that occurs if at least one of the participants starts to deviate from the honest behavior. In the optimistic case, the protocols proceed off-chain until the final payout needs to be enforced. This case is both the cheapest and fastest possible outcome of the protocol. If an honest party realizes that another party misbehaves, i.e., deviates from honest behavior, it starts a dispute. The parties have no chance to resolve such a disagreement off-chain. Therefore they need the blockchain as a trusted judge that solves the dispute (based on the rules of the protocol and the inputs of the disagreeing parties). This scenario can, in the very worst case, be slower and more expensive than the direct on-chain evaluation. But we showed for each protocol that no party can (financially) benefit from

¹In fact, there could be more than one pessimistic case, but we only consider the worst possible scenario here.

7 Conclusion

misbehaving in any way. Therefore, we assume that usually, parties will behave honestly to avoid transaction fees in the dispute procedure.

For all three protocols, we analyzed the security through precise cryptographic proofs and evaluated their efficiency through a proof-of-concept implementation and benchmarks.

In the PERUN protocol (cf. Chapter 4), we only focus on payments and show that a single on-chain setup with a hub suffices to support off-chain payments between many users. We presented the concept of virtual payment channels, which can be built on top of two ledger channels. While the concept of payment channels was already known, the novelty of this construction is the introduction of virtual channels that can be opened and closed off-chain (in the optimistic case). In contrast to existing proposals of payment routing, virtual channels have the advantage that they support payments without interaction with the intermediary, that connects the two ledger channels. We evaluated the security of the PERUN protocol in the UC framework and showed that it satisfies an idealized protocol version (cf. Section 4.3). In particular, it guarantees that all parties agree off-chain about the current state of the channels and that no coins can be stolen or lost.

In two extensions to the PERUN paper [69], the concept of virtual channels was also applied to off-chain smart contract execution between two parties [71] and n parties [68]. The result is a complex framework that does not only scale payment systems but also the execution of complex contracts. However, as malicious parties could always potentially force the on-chain evaluation, this system only scales contracts that can already be run in cryptocurrencies like Ethereum. The FAIRSWAP and FASTKITTEN protocols support contracts with features that go beyond this case. In fact, we showed how powerful complex functions could be evaluated off-chain where even in the pessimistic case, the on-chain computation stays small.

The FAIRSWAP protocol (cf. Chapter 5) allows the fair sale of very large digital goods or witness x , where a smart contract acts as a judge and verifies their correctness using concise proofs of misbehavior. These proofs are short statements that are generated by the receiver if the witness x does not satisfy a (potentially very complex) verification circuit ϕ . An advantage of the FAIRSWAP protocol is that it supports very large witnesses x and highly complex functions ϕ . In fact, it even supports files so large that storing them in Ethereum could exceed the gas limit of blocks in the blockchain. Therefore FAIRSWAP is able to support applications that would otherwise be infeasible on Ethereum. We show that the protocol terminates in at most 5Δ rounds (where Δ is the blockchain delay) and that the

7 Conclusion

costs can be kept very low, both in the optimistic but also in the pessimistic case.

FAIRSWAP is focused on the evaluation of a single function between two participants, the FASTKITTEN protocol takes a more general approach. It utilizes a single TEE to support the efficient off-chain evaluation of *generic* smart contract between a fixed set of parties. In Chapter 6, we have shown that such efficient off-chain smart contracts are even possible using only standard transactions by combining blockchain technology with trusted hardware. We constructed the FASTKITTEN protocol, a Bitcoin compatible smart contract execution framework which supports efficient multi-round contracts. We show that the system is highly practical as it enables real-time application scenarios, like interactive online gaming, with millisecond round latencies between participants (in the optimistic case). We show that the protocol achieves a high level of security for the protocol participants as well as the operator of the TEE. Additionally, we discuss multiple extensions to our protocol, such as adding output privacy or operator fees, which enrich the set of features provided by our system.

While PERUN and FAIRSWAP require a more advanced scripting language, i.e., smart contracts, FASTKITTEN runs on top of standard blockchain instructions as provided by Bitcoin. Nevertheless, all three protocols are blockchain agnostic, which means they are not specifically built for these two cryptocurrencies but can be supported by any permissioned or permissionless blockchain with similar features. In particular, we abstract from the exact specification of a blockchain and rely only upon the *blockchain assumption*, which states that the underlying ledger provides liveness and consistency (cf. Section 3.4). In particular, we assume that the blockchain does not get forked or congested longer than Δ rounds) to prevent that transactions of honest parties get accepted. It is possible to mitigate the risk of forks and congestion by using high transaction fees by default and set a very long timeout parameter (Δ). However, both measures make our protocols less efficient. In practice, it might make sense to try a more dynamic approach, i.e., increase transaction fees and timelock parameters ad-hoc, e.g., when blockchain congestion occurs. It might also be useful to adapt these measures based on the overall transaction amounts. If the protocol only considers a few euros, many users might want to risk the trade-off of reduced security for higher efficiency. But if many thousand euros are at stake, a timeout of a few hours up to a day is a small price to pay for strong security.

Future Work One further research directions is a detailed analysis of the privacy of the presented schemes. It would be interesting to understand if PERUN and

7 Conclusion

FAIRSWAP could be extended to provide privacy of transactions towards third parties or intermediaries. However, even defining privacy is non-trivial for corrupted parties, as participants need full knowledge of the values. Another interesting open question is whether the presented techniques could also work on privacy-preserving blockchain technologies and help these platforms to scale securely.

Another promising research direction is to investigate how the scaling effect can be amplified even further. As this work focuses on increasing the scalability on the application layer only, it does not analyze the scaling effect of running the protocols on top of consensus-layer scaling techniques. Additionally, different application-layer techniques could be combined, e.g., to reduce the collateral costs or gas fees. While this looks like a promising solution to reach much higher scalability, it remains to show how these changes impact the protocols' security.

All three presented protocols could also benefit from additional game-theoretic analysis. Research on the strategies of incentive-driven adversaries can lead to a better understanding of the protocols' economic security. Identifying dominant strategies would also help in estimating the execution costs for the protocols. As the optimal outcome would be that the equilibrium of each protocol is the optimistic case, a detailed analysis can help identify further incentive mechanisms for a fast and cheap protocol evaluation.

In summary, all three presented protocols improve blockchain scalability by scaling the applications that run on top of them. FAIRSWAP and PERUN have both been analyzed and extended by follow up works. The protocols discussed in this thesis show that off-chain protocols are a promising research direction, and further results in this area are required to make blockchain technology suitable for mass adoption.

List of Figures

3.1	A simple transaction tx with a single input and output.	25
3.2	Bitcoin Blockchain	26
4.1	Ledger channel β between end-parties \mathcal{A} and \mathcal{B}	49
4.2	Setup for payment routing from Alice to Bob over Ingrid.	50
4.3	Setup for a virtual channel γ between Alice and Bob.	52
4.4	Ledger channels $\beta_{\mathcal{A}}$ and $\beta_{\mathcal{B}}$ after closing γ	53
4.5	Message flow between Alice, Bob and Ingrid during opening of γ	79
4.6	Message flow between Alice, Bob and Ingrid during closing of γ	83
4.7	Setup of a simulation of the PERUN security proof in UC-style manner for honest parties.	87
5.1	Setup for digital fair sale between sender \mathcal{S} and receiver \mathcal{R}	104
5.2	Naïve smart contract based fair sale.	106
5.3	Zero-Knowledge Contingent Payment (ZKCP) based fair sale.	107
5.4	The FAIRSWAP protocol for fair sale	108
5.5	Outline of fair exchange with judge contract	125
5.6	Setup of a Simulation with honest parties	135
5.7	Simulation with corrupted sender \mathcal{S}^* and honest receiver \mathcal{R}	139
5.8	Simulation against \mathcal{Z} with honest sender \mathcal{S} and malicious receiver \mathcal{R}^*	143
5.9	Merkle tree circuit for exchange of $x = (x_1, \dots, x_8)$	151
5.10	Costs and encoding size for different values of λ	154
6.1	Setup and communication of FASTKITTEN protocol.	164
6.2	FASTKITTEN Protocol	177
6.3	Deposit transactions issued by the operator \mathcal{Q} during the Setup phase.	179
6.4	Penalty transactions issued by the enclave during the setup.	179
6.5	Challenge response transactions for the i -th round.	181
6.6	Output transaction issued by the FASTKITTEN enclave.	182

List of Figures

List of Tables

3.1	Average time and number of blocks that it takes until a Bitcoin transaction is included (numbers from [27])	28
3.2	Average time and number of blocks that it takes until an Ethereum transaction is processed.	31
4.1	PERUN execution fees (with exchange rates from Section 3.2.1). . .	97
5.1	Gas cost comparison between FairSwap, SmartJudge, and OptiSwap. Numbers taken from related work (with exchange rates from Section 3.2.1).	157
6.1	Selected solutions for contract execution over Bitcoin and their comparison to Ethereum smart contracts. Above, n denotes the number of parties and m is the number of reactive execution rounds. . . .	171
6.2	Estimated fees for a typical deposit transaction and the FASTKIT-TEN transactions.	187

List of Tables

List of Abbreviations

CRS	Common Reference String
DAG	Directed Acyclic Graph
DoS	Denial of Service
ECDSA	Elliptic Curve Digital Signature Algorithm
EVM	Ethereum Virtual Machine
GUC	Global UC
HTLC	Hashed Time Locked Contract
IoT	Internet of Things
MPC	Multi-Party Computation
NIZK	Non-interactive Zero-Knowledge Proof
PoS	Proof of Stake
PoW	Proof of Work
PPT	Probabilistic Polynomial Time
ROM	Random Oracle Model
SGX	Software Guard Extension
SCRIPT	Bitcoin scripting language
SHA-3	Secure Hash Algorithm 3
TEE	Trusted Execution Environment
TPS	Transaction per Second
TTP	Trusted Third Party
UC	Universally Composable
UTXO	Unspent Transaction Output
ZKCP	Zero-Knowledge Contingent Payment

List of Abbreviations

Bibliography

- [1] 1ML. *Lightning Network Statistics: Lightning Network Search and Analysis Engine - Bitcoin mainnet*. URL: <https://1ml.com/statistics> (visited on 02/15/2020).
- [2] ACINQ/eclair. URL: <https://github.com/ACINQ/eclair> (visited on 02/15/2020).
- [3] E. Adar and B. A. Huberman. “Free riding on Gnutella”. In: *First monday* 10 (2000).
- [4] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. “Innovative Technology for CPU Based Attestation and Sealing”. In: *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.
- [5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. “Secure Multiparty Computations on Bitcoin”. In: *Proceedings of the 21st ACM Conference on Computer and Communications Security*. 2014, pp. 443–458.
- [6] AORA. *Shipping from US to Singapore*. URL: <https://www.aora.com/> (visited on 01/08/2020).
- [7] ARM Limited. “Security technology-building a secure system using trustzone technology”. In: *ARM Technical White Paper* (2009). URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C.trustzone_security_whitepaper.pdf.
- [8] N. Asokan, M. Schunter, and M. Waidner. “Optimistic protocols for fair exchange”. In: *Proceedings of the 4th ACM conference on Computer and communications security*. 1997, pp. 7–17.
- [9] N. Atzei, M. Bartoletti, and T. Cimoli. *A survey of attacks on Ethereum smart contracts: Cryptology ePrint Archive, Report 2016/1007*. 2016.
- [10] Aurora Labs. *IDEX - Decentralized Ethereum Asset Exchange*. 2019. URL: <https://idex.market/static/IDEX-Whitepaper-V0.7.6.pdf> (visited on 01/08/2020).
- [11] G. Avarikioti, O. S. T. Litos, and R. Wattenhofer. “Cerberus Channels: Incentivizing Watchtowers for Bitcoin”. In: *Financial Cryptography and Data Security (FC)* (2020).
- [12] A. Back. *Hashcash-a denial of service counter-measure*. 2002. URL: <http://www.hashcash.org/papers/hashcash.pdf> (visited on 02/15/2020).

Bibliography

- [13] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. “Bitcoin as a Transaction Ledger: A Composable Treatment”. In: *Annual International Cryptology Conference*. 2017, pp. 324–356.
- [14] W. Banasik, S. Dziembowski, and D. Malinowski. “Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II”. In: *ESORICS 2016 - 21st European Symposium on Research in Computer Security*. Heraklion, Greece, 2016, pp. 261–280.
- [15] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. “Consensus in the age of blockchains”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019.
- [16] J. Barbie. *Why Smart Contracts are NOT feasible on Plasma*. EthResearch.ch, 2018. URL: <https://ethresearch.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598> (visited on 02/15/2020).
- [17] B. Barhydt. *Crypto Bites: Chat with Ethereum founder Vitalik Buterin*. youtube.com, 2019. URL: https://youtu.be/u-i_mTwL-FI (visited on 01/28/2020).
- [18] O. Barzilay. *Why Blockchain Is The Future Of The Sharing Economy*. www.forbes.co, 2017. URL: <https://www.forbes.com/sites/omribarzilay/2017/08/14/why-blockchain-is-the-future-of-the-sharing-economy/#623fb4ac3342> (visited on 10/08/2020).
- [19] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. “Chainspace: A Sharded Smart Contracts Platform”. In: *25th Annual Network and Distributed System Security Symposium, NDSS*. February 18-21, 2018.
- [20] M. Bellare, G. Fuchsbauer, and A. Scafuro. “NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion”. In: *22nd International Conference on the Theory and Application of Cryptology and Information Security*. Hanoi, Vietnam, 2016, pp. 777–804.
- [21] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable Zero Knowledge via Cycles of Elliptic Curves”. In: *Advances in cryptology - CRYPTO 2014*. Santa Barbara, CA, USA, 2014, pp. 276–294.
- [22] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, and A. Juels. “Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware”. In: *IACR Cryptology ePrint Archive (2017)*. URL: <http://eprint.iacr.org/2017/1153>.
- [23] I. Bentov and R. Kumaresan. “How to Use Bitcoin to Design Fair Protocols”. In: *Advances in cryptology - CRYPTO 2014*. Santa Barbara, CA, USA, 2014, pp. 421–439.

Bibliography

- [24] A. Bhuptani, L. Haber, R. Sethuram, and H. Hillman. *Connex Network*. 2019. URL: <https://connex.network/> (visited on 12/30/2019).
- [25] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel 5SGX6”. In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, USA, 2018, pp. 1213–1227.
- [26] *Bitcoin - Google Scholar: Citation count for Bitcoin whitepaper*. 2020. URL: <https://scholar.google.de/scholar?q=Bitcoin> (visited on 01/11/2020).
- [27] *Bitcoin Fees for Transactions*. 2020. URL: <https://bitcoinfees.earn.com/> (visited on 02/20/2020).
- [28] blockchain.com. *BTC auf USD: Bitcoin auf US-Dollar Marktpreis - Blockchain*. 2020. URL: <https://www.blockchain.com/de/charts/market-price?timespan=180days> (visited on 02/25/2020).
- [29] M. Blum, P. Feldman, and S. Micali. “Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract)”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 103–112.
- [30] R. Böhme, L. Eckey, N. Narula, T. Moore, T. Ruffing, and A. Zohar. “Responsible Vulnerability Disclosure in Cryptocurrencies”. In: *Communications of the ACM (CACM)*. 10. 2020, pp. 62–71.
- [31] M. Bowman, A. Miele, M. Steiner, and B. Vavala. “Private data objects: an overview”. In: *ArXiv e-prints* (2018). URL: <https://arxiv.org/abs/1807.05686> (visited on 02/15/2020).
- [32] S. Brands. “Untraceable off-line cash in wallet with observers”. In: *Annual international cryptology conference*. 1993, pp. 302–318.
- [33] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostianen, U. Müller, and A.-R. Sadeghi. “DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization”. In: *Proceedings of the 35th Annual Computer Security Applications Conference*. 2019, pp. 788–800.
- [34] F. Brasser, U. Müller, A. Dmitrienko, K. Kostianen, S. Capkun, and A.-R. Sadeghi. “Software grand exposure: SGX cache attacks are practical”. In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, BC, Canada, 2017.
- [35] C. Burchert, C. Decker, and R. Wattenhofer. “Scalable funding of bitcoin micro-payment channel networks”. In: *Royal Society open science* 8 (2018), p. 180089.
- [36] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. “Control-Flow Integrity: Precision, Security, and Performance”. In: *CoRR* (2016).

Bibliography

- [37] V. Buterin. *A next-generation smart contract and decentralized application platform*. 2014. URL: <https://github.com/ethereum/wiki/wiki/White-Paper> (visited on 02/15/2020).
- [38] V. Buterin. *Minimal Viable Plasma: Ethereum Research Forum*. 2018. URL: <https://ethresear.ch/t/minimal-viable-plasma/426> (visited on 02/22/2020).
- [39] V. Buterin and V. Griffith. *Casper the friendly finality gadget*. 2017. URL: <https://arxiv.org/abs/1710.09437> (visited on 02/15/2020).
- [40] C. Cachin and J. Camenisch. “Optimistic Fair Secure Computation”. In: *Advances in Cryptology - CRYPTO 2000*. Santa Barbara, CA, USA, 2000, pp. 93–111.
- [41] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. “The Wonderful World of Global Random Oracles”. In: *Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Tel Aviv, Israel, 2018, pp. 280–312.
- [42] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo. “Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services”. In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. Dallas, TX, USA, 2017, pp. 229–243.
- [43] R. Canetti. “Security and Composition of Multiparty Cryptographic Protocols”. In: *Journal of Cryptology* 1 (2000), pp. 143–202.
- [44] R. Canetti. “Universally composable security: a new paradigm for cryptographic protocols”. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 2001, pp. 136–145.
- [45] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. “Universally composable security with global setup”. In: *Theory of Cryptography Conference*. Berlin, 2007, pp. 61–85.
- [46] R. Canetti, A. Jain, and A. Scafuro. “Practical UC security with a Global Random Oracle”. In: *Proceedings of the 21st ACM Conference on Computer and Communications Security*. 2014, pp. 597–608.
- [47] R. Canetti, B. Riva, and G. N. Rothblum. “Practical delegation of computation using multiple servers”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. Chicago, Illinois, USA, 2011, pp. 445–454.
- [48] M. Castro, B. Liskov, et al. “Practical Byzantine Fault Tolerance”. In: *Third Symposium on Operating Systems Design and Implementation*. 1999, pp. 173–186.
- [49] D. Chaum. “Security without identification: Transaction systems to make big brother obsolete”. In: *Communications of the ACM* 10 (1985), pp. 1030–1044.

Bibliography

- [50] D. Chaum, A. Fiat, and M. Naor. “Untraceable electronic cash”. In: *Conference on the Theory and Application of Cryptography*. 1988, pp. 319–327.
- [51] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. “Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017.
- [52] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts”. In: *4th IEEE European Symposium on Security and Privacy*. Stockholm, Sweden, 2019, pp. 185–200.
- [53] A. Chiesa, E. Tromer, and M. Virza. “Cluster Computing in Zero Knowledge”. In: *34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Sofia, Bulgaria, 2015, pp. 371–403.
- [54] N. Christin. “Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace”. In: *Proceedings of the 22nd international conference on World Wide Web*. 2013, pp. 213–224.
- [55] *CoinMarketCap*. 2020. URL: <https://coinmarketcap.com/> (visited on 02/15/2020).
- [56] J. Coleman, L. Horne, and L. Xuanji. *Counterfactual: Generalized State Channels*. 2018. URL: <https://14.ventures/papers/statechannels.pdf> (visited on 02/15/2020).
- [57] V. Costan, I. A. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, USA, 2016.
- [58] K. Croman et al. “On Scaling Decentralized Blockchains”. In: *Financial Cryptography and Data Security*. 2016, pp. 106–125.
- [59] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A.-R. Sadeghi. “FastKitten: Practical Smart Contracts on Bitcoin”. In: *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA, USA, 2019, pp. 801–818.
- [60] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. “Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM”. In: *7th International Workshop on Virtualization Technologies in Distributed Computing*. 2013.
- [61] C. Decker and R. Wattenhofer. “A fast and scalable payment network with bitcoin duplex micropayment channels”. In: *Symposium on Self-Stabilizing Systems*. 2015, pp. 3–18.

Bibliography

- [62] G. Di Stasi, S. Avallone, R. Canonico, and G. Ventre. “Routing payments on the lightning network”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2018, pp. 1161–1170.
- [63] dice2.win. *Fair games that pay Ether*. URL: <https://dice2.win/> (visited on 01/08/2020).
- [64] Digiconomist. *Bitcoin Energy Consumption Index*. 2020. URL: <https://digiconomist.net/bitcoin-energy-consumption> (visited on 01/16/2020).
- [65] J. R. Douceur. “The sybil attack”. In: *International workshop on peer-to-peer systems*. 2002, pp. 251–260.
- [66] A. D. Dwivedi. *A Scalable Blockchain Based Digital Rights Management System*. 2019. URL: <https://eprint.iacr.org/2019/1217.pdf> (visited on 02/15/2020).
- [67] S. Dziembowski, L. Eckey, and S. Faust. “FairSwap: How To Fairly Exchange Digital Goods”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada, 2018, pp. 967–984.
- [68] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. “Multi-party Virtual State Channels”. In: *Proceedings of the 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Darmstadt, Germany, 2019, pp. 625–656.
- [69] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. “Perun: Virtual Payment Hubs Over Cryptocurrencies”. In: *Proceedings of the 40th IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 2019, pp. 327–344.
- [70] S. Dziembowski, G. Fabiański, S. Faust, and S. Riahi. *Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma*. eprint, 2020. URL: ia.cr/2020/175 (visited on 02/21/2020).
- [71] S. Dziembowski, S. Faust, and K. Hostáková. “General state channel networks”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada, 2018, pp. 949–966.
- [72] L. Eckey, S. Faust, and J. Loss. “Efficient Algorithms for Broadcast and Consensus Based on Proofs of Work”. In: *IACR Cryptology ePrint Archive (2017)*. URL: <https://eprint.iacr.org/2017/915>.
- [73] L. Eckey, S. Faust, and B. Schlosser. “OptiSwap: Fast Optimistic Fair Exchange”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. Taipei, Taiwan, 2020, pp. 543–557.

Bibliography

- [74] C. Egger, P. Moreno-Sanchez, and M. Maffei. “Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks”. In: *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*. London, UK, 2019, pp. 801–815.
- [75] *ElementsProject/lightning*. URL: <https://github.com/ElementsProject/lightning> (visited on 02/15/2020).
- [76] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. *A²L: Anonymous Atomic Locks for Scalability and Interoperability in Payment Channel Hubs*. Cryptology ePrint Archive, Report 2019/589. 2019.
- [77] Ú. Erlingsson and J. Ligatti. “Control-flow integrity principles, implementations, and applications”. In: *ACM Transactions on Information System Security* (2009).
- [78] Etherscan.io. *Ether Daily Price (USD) Chart — Etherscan*. 2020. URL: <https://etherscan.io/chart/etherprice> (visited on 02/25/2020).
- [79] M. Fischlin, A. Lehmann, T. Ristenpart, T. Shrimpton, M. Stam, and S. Tessaro. “Random Oracles with(out) Programmability”. In: *ASIACCS ’10: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. 2010, pp. 303–320.
- [80] E. Foundation. *Ethereum Constantinople Upgrade Announcement*. 2019. URL: <https://blog.ethereum.org/2019/01/11/ethereum-constantinople-upgrade-announcement/> (visited on 02/20/2020).
- [81] G. Fuchsbauer. “WI Is Not Enough”. In: *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*. London, UK, 2019, pp. 49–62.
- [82] Fun Fair Technologies. *Blockchain Solutions for Gaming: Commercial Whitepaper: v 2.0*. 2018. URL: <https://funfair.io/wp-content/uploads/FunFair-Commercial-White-Paper-v2-draft.pdf> (visited on 01/08/2020).
- [83] H. Galal and A. Youssef. “Verifiable sealed-bid auction on the ethereum blockchain”. In: *Financial Cryptography and Data Security*. Nieuwpoort, Curaçao, 2019.
- [84] J. A. Garay, A. Kiayias, and N. Leonardos. “The Bitcoin Backbone Protocol with Chains of Variable Difficulty”. In: *Advances in Cryptology – CRYPTO 2017*. Santa Barbara, CA, USA, 2017.
- [85] J. Garay, A. Kiayias, and N. Leonardos. “The bitcoin backbone protocol: Analysis and applications”. In: *Advances in cryptology - CRYPTO 2015*. Santa Barbara, CA, USA, 2015, pp. 281–310.
- [86] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. “Brick: Asynchronous State Channels”. In: *CoRR* (2019).

Bibliography

- [87] I. Giacomelli, J. Madsen, and C. Orlandi. “ZKBoo: Faster Zero-Knowledge for Boolean Circuits”. In: *Proceedings of the 25th USENIX Security Symposium*. Austin, TX, USA, 2016, pp. 1069–1083.
- [88] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. “Algorand: Scaling byzantine agreements for cryptocurrencies”. In: *SOSP’17*. 2017, pp. 51–68.
- [89] O. Goldreich. *Foundations of Cryptography: Volume 1*. New York, NY, USA, 2006.
- [90] O. Goldreich, S. Micali, and A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pp. 218–229.
- [91] M. Green and I. Miers. “Bolt: Anonymous payment channels for decentralized currencies”. In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. Dallas, TX, USA, 2017, pp. 473–489.
- [92] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory”. In: *Proceedings of the 26th USENIX Security Symposium*. Vancouver, BC, Canada, 2017.
- [93] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. “SoK: Off The Chain Transactions”. In: *IACR Cryptology ePrint Archive* (2019), p. 360.
- [94] M. Hachman. *Intel’s plan to fix Meltdown in silicon raises more questions than answers*. 2018.
- [95] M. Hearn. *Contract: Example 7: Rapidly adjusted (micro)payments to a pre-determined party: [Bitcoin Wiki]*. Bitcoin Wiki, 2011. URL: <https://en.bitcoin.it/w/index.php?title=Contract&oldid=20401>.
- [96] M. Hearn. *Micro-payment channels implementation now in bitcoinj*. Bitcointalk.org, 2013. URL: <https://bitcointalk.org/index.php?topic=244656.0>.
- [97] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. “TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub”. In: *24th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA, 2017.
- [98] K. Hickman and T. Elgamal. “The SSL protocol”. In: (1995).
- [99] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. “Using Innovative Instructions to Create Trustworthy Software Solutions”. In: *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

Bibliography

- [100] Intel. *Intel Software Guard Extensions Developer Guide*. 2016. URL: https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf (visited on 02/25/2020).
- [101] Intel. *Resources and Response to Side Channel L1 Terminal Fault*. 2018. URL: <https://www.intel.com/content/www/us/en/architecture-and-technology/11tf.html> (visited on 02/25/2020).
- [102] M. Jawurek, F. Kerschbaum, and C. Orlandi. “Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently”. In: *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*. Berlin, Germany, 2013, pp. 955–966.
- [103] M. Jourenko, K. Kurazumi, M. Larangeira, and K. Tanaka. “SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies”. In: *IACR Cryptology ePrint Archive* (2019), p. 352.
- [104] A. Juels, A. E. Kosba, and E. Shi. “The Ring of Gyges: Investigating the Future of Criminal Smart Contracts”. In:
- [105] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. “Arbitrum: Scalable, private smart contracts”. In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, USA, 2018, pp. 1353–1370.
- [106] G. Kaptchuk, I. Miers, and M. Green. “Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers”. In: *26th Annual Network and Distributed System Security Symposium, NDSS*. San Diego, California, USA, 2019.
- [107] M. Karakaya, I. Körpeoğlu, and Ö. Ulusoy. “Counteracting free riding in Peer-to-Peer networks”. In: *Computer Networks* 3 (2008), pp. 675–694.
- [108] J. Katz and Y. Lindell. *Introduction to modern cryptography*. Boca Raton, Fla., 2008.
- [109] J. Katz, U. Maurer, B. Tackmann, and V. Zikas. “Universally Composable Synchronous Computation”. In: *tcc13name*. 2013, pp. 477–498.
- [110] M. Khabbazian, T. Nadahalli, and R. Wattenhofer. “Outpost: A responsive lightweight watchtower”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019, pp. 31–40.
- [111] R. Khalil and A. Gervais. “NOCUST-A Non-Custodial 2nd-Layer Financial Intermediary”. In: *IACR Cryptology ePrint Archive* (2018), p. 642.
- [112] R. Khalil and A. Gervais. “Revive: Rebalancing off-blockchain payment networks”. In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. Dallas, TX, USA, 2017, pp. 439–453.

Bibliography

- [113] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. “Cutting the gordian knot: A look under the hood of ransomware attacks”. In: *Advances in cryptology - CRYPTO 2015*. Santa Barbara, CA, USA, 2015, pp. 3–24.
- [114] A. Kiayias and O. S. T. Litos. “A Composable Security Treatment of the Lightning Network”. In: *Cryptology ePrint Archive*.
- [115] A. Kiayias and G. Panagiotakos. “Speed-Security Tradeoffs in Blockchain Protocols”. In: *IACR Cryptology ePrint Archive (2015)*, p. 1019.
- [116] A. Kiayias, A. Russell, B. David, and R. Oliynykov. “Ouroboros: A provably secure proof-of-stake blockchain protocol”. In: *Annual International Cryptology Conference*. 2017, pp. 357–388.
- [117] A. Kiayias, H.-S. Zhou, and V. Zikas. “Fair and Robust Multi-Party Computation using a Global Transaction Ledger”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt) (2016)*, pp. 705–734.
- [118] U. Klarman, S. Basu, A. Kuzmanovic, and E. G. Sirer. “bloxroute: A scalable trustless blockchain distribution network whitepaper”. In: *IEEE Internet of Things Journal* (2018).
- [119] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. “Omiledger: A secure, scale-out, decentralized ledger via sharding”. In: *39th IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 2018, pp. 583–598.
- [120] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts”. In: *37th IEEE Symposium on Security and Privacy*. San Jose, California, USA, 2016, pp. 839–858.
- [121] R. Kumaresan and I. Bentov. “Amortizing Secure Computation with Penalties”. In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016, pp. 418–429.
- [122] R. Kumaresan and I. Bentov. “How to use bitcoin to incentivize correct computations”. In: *Proceedings of the 21st ACM Conference on Computer and Communications Security*. 2014.
- [123] R. Kumaresan, T. Moran, and I. Bentov. “How to use bitcoin to play decentralized poker”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Denver, CO, USA, 2015.

Bibliography

- [124] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. “Improvements to Secure Computation with Penalties”. In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016, pp. 406–417.
- [125] A. Küpçü and A. Lysyanskaya. “Usable optimistic fair exchange”. In: *Computer Networks* 1 (2012), pp. 50–63.
- [126] Kyber Network. *Kyber: An On-Chain Liquidity Protocol: v 0.1*. 2019. URL: https://files.kyber.network/Kyber_Protocol_22_April_v0.1.pdf (visited on 01/08/2020).
- [127] L. Lamport, R. Shostak, and M. Pease. “The Byzantine generals problem”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).
- [128] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. “SoK: Automated Software Diversity”. In: *35th IEEE Symposium on Security and Privacy*. San Jose, California, USA, 2014.
- [129] M. Lewis. *Multimillionaire 25-year-old crypto king Vitalik Buterin speaks to the Star about the future of Ethereum* —. 2019. URL: <https://www.thestar.com/business/2019/08/19/ethereums-vitalik-buterin-on-reducing-cryptocurrency-risks.html> (visited on 01/08/2020).
- [130] *lightningnetwork/lnd*. URL: <https://github.com/lightningnetwork/lnd> (visited on 02/15/2020).
- [131] J. Lind, O. Naor, I. Eyal, F. Kelbert, P. R. Pietzuch, and E. G. Sirer. “Teechain: Reducing Storage Costs on the Blockchain With Offline Payment Channels”. In: *Proceedings of the 11th ACM International Systems and Storage Conference*. 2018, p. 125.
- [132] E. Lombrozo, J. Lau, and P. Wuille. *Bitcoin Improvement Proposal (BIP) 414: Segregated Witness (Consensus layer)*. 2017. URL: <https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki> (visited on 01/08/2020).
- [133] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. “A Secure Sharding Protocol For Open Blockchains”. In: pp. 17–30. URL: <http://doi.acm.org/10.1145/2976749.2978389>.
- [134] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei. “SilentWhispers: Enforcing Security and Privacy in Decentralized Credit Networks”. In: *24th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA, 2017.
- [135] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. “Concurrency and privacy with payment-channel networks”. In: *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*. Dallas, TX, USA, 2017, pp. 455–471.

Bibliography

- [136] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. “Multi-Hop Locks for Secure, Privacy-Preserving and Interoperable Payment-Channel Networks”. In: *IACR Cryptology ePrint Archive* (2018), p. 472.
- [137] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. “Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings”. In: London, United Kingdom, 2019, pp. 2111–2128.
- [138] Matt Corallo. *FIBRE Fast Internet Bitcoin Relay Engine*. 2019. URL: <https://bitcoinfibre.org/> (visited on 01/30/2020).
- [139] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle. “A Quantitative Analysis of the Impact of Arbitrary Blockchain Content on Bitcoin”. In: *Financial Cryptography and Data Security*. Nieuwpoort, Curaçao, 2019.
- [140] G. Maxwell. *The first successful Zero-Knowledge Contingent Payment*. bitcoincore.org, 2016. URL: <https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/> (visited on 01/16/2020).
- [141] P. McCorry, S. Bakshi, I. Bentov, A. Miller, and S. Meiklejohn. “Pisa: Arbitration Outsourcing for State Channels”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019.
- [142] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative Instructions and Software Model for Isolated Execution”. In: *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.
- [143] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. “A fistful of bitcoins: characterizing payments among men with no names”. In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013, pp. 127–140.
- [144] R. C. Merkle. *Secrecy, authentication, and public key systems*. 1979.
- [145] C. Meyer and J. Schwenk. “SoK: Lessons learned from SSL/TLS attacks”. In: *International Workshop on Information Security Applications*. 2013, pp. 189–209.
- [146] I. Miers, C. Garman, M. Green, and A. D. Rubin. “ZeroCoin: Anonymous distributed e-cash from bitcoin”. In: *34th IEEE Symposium on Security and Privacy*. San Francisco, California, USA, 2013, pp. 397–411.
- [147] A. Miller and I. Bentov. “Zero-collateral lotteries in Bitcoin and Ethereum”. In: *2nd IEEE European Symposium on Security and Privacy*. Paris, France, 2017.

Bibliography

- [148] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry. “Sprites: Payment Channels that Go Faster than Lightning”. In: *Financial cryptography and data security*. Frigate Bay, St. Kitts and Nevis, 2019.
- [149] A. Miller, A. Juels, E. Shi, B. J. Parno, and J. Katz. “Permacoin: Repurposing bitcoin work for data preservation”. In: *IEEE Security & Privacy* (2014), pp. 475–490.
- [150] S. Nakamoto. *Bitcoin P2P e-cash paper*. The Cryptography Mailing List, 2008. URL: <https://www.mail-archive.com/cryptography@metzdowd.com/msg09959.html> (visited on 01/27/2020).
- [151] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009. URL: <http://bitcoin.org/bitcoin.pdf> (visited on 02/15/2020).
- [152] J. B. Nielsen. “Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case”. In: *Advances in Cryptology - CRYPTO 2002*. Santa Barbara, CA, USA, 2002, pp. 111–126.
- [153] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. “Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base”. In: *Proceedings of the 22nd USENIX Security Symposium*. 2013.
- [154] OECD. *Gross domestic product (GDP) and spending: GDP indicator*. 2020. URL: <https://data.oecd.org/gdp/gross-domestic-product-gdp.htm> (visited on 01/27/2019).
- [155] H. Pagnia and F. C. Gärtner. *On the impossibility of fair exchange without a trusted third party*. 1999.
- [156] Parity Technologies. *The Multi-sig Hack: A Postmortem*. 2017. URL: <https://www.parity.io/the-multi-sig-hack-a-postmortem/> (visited on 02/15/2020).
- [157] B. J. Parno, J. Howell, C. Gentry, and M. Raykova. “Pinocchio: Nearly Practical Verifiable Computation”. In: *34th IEEE Symposium on Security and Privacy*. San Francisco, California, USA, 2013, pp. 238–252.
- [158] R. Pass, L. Seeman, and A. Shelat. “Analysis of the blockchain protocol in asynchronous networks”. In: *Annual International Cryptology Conference*. 2017, pp. 643–673.
- [159] R. Pass, E. Shi, and F. Tramèr. “Formal Abstractions for Attested Execution Secure Processors”. In: *36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Paris, France, 2017.
- [160] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. 2017. URL: [Plasma,%20https://plasma.io/plasma.pdf/](https://plasma.io/plasma.pdf).

Bibliography

- [161] J. Poon and T. Dryja. *The bitcoin lightning network: Scalable off-chain instant payments*. 2016. URL: <https://lightning.network/lightning-network-paper.pdf>.
- [162] S. Popov. “The Tangle: Version 1.4.3”. 2018. URL: https://iota.org/IOTA_Whitepaper.
- [163] P. Prihodko, S. Zhigulin, M. Sahno, A. Ostrovskiy, and O. Osuntokun. “Flare: An approach to routing in lightning network”. In: *White paper* (2016).
- [164] M. R. Rahman. “A survey of incentive mechanisms in peer-to-peer systems”. In: *Cheriton School of Computer Science, University of Waterloo, Tech. Rep. CS-2009-22* (2009).
- [165] Raiden Network. *Raiden Protocol Explained*. 2019. URL: <https://raiden.network/101.html> (visited on 02/15/2020).
- [166] L. Ramaswamy and L. Liu. “Free riding: A new challenge to peer-to-peer file sharing systems”. In: *Proceedings of the 36th Annual International Conference on System Sciences*. Big Island, Hawaii, 2003, 10–pp.
- [167] P. Rogaway. “Formalizing human ignorance”. In: *International Conference on Cryptology in Vietnam*. 2006, pp. 211–228.
- [168] E. Rohrer and F. Tschorsch. “Kadcast”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019, pp. 199–213.
- [169] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. “Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions”. In: *25th Annual Network and Distributed System Security Symposium, NDSS*. February 18–21, 2018, pp. 455–471.
- [170] T. Ruffing, S. A. Thyagarajan, V. Ronge, and D. Schroder. “(Short Paper) Burning Zerocoins for Fun and for Profit-A Cryptographic Denial-of-Spending Attack on the Zerocoin Protocol”. In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2018, pp. 116–119.
- [171] E. B. Sasse, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized anonymous payments from bitcoin”. In: *35th IEEE Symposium on Security and Privacy*. San Jose, California, USA, 2014, pp. 459–474.
- [172] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. “SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs”. In: *24th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA, 2017.

Bibliography

- [173] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. In: *24th Annual Network and Distributed System Security Symposium*. San Diego, CA, USA, 2017.
- [174] *Slock.it*. URL: <https://slock.it> (visited on 01/08/2020).
- [175] Y. Sompolinsky and A. Zohar. *Accelerating Bitcoin’s Transaction Processing. Fast Money Grows on Trees, Not Chains*. 2013. URL: <https://eprint.iacr.org/2013/881> (visited on 02/15/2020).
- [176] J. Spilman. *PaymentChannel.cs*. Github.com, 2013. URL: <https://gist.github.com/jspilman/5424310> (visited on 02/15/2020).
- [177] J. Teutsch and C. Reitwießner. *TrueBit – a scalable verification solution for blockchains*. 2018. URL: <https://arxiv.org/abs/1908.04756> (visited on 02/15/2020).
- [178] P. Todd. *BIP 65: OP_CHECKLOCKTIMEVERIFY*. 2014. URL: <https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki> (visited on 02/20/2020).
- [179] F. Tramèr, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi. “Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge”. In: *2nd IEEE European Symposium on Security and Privacy*. Paris, France, 2017.
- [180] J. van Bulck, F. Piessens, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. Baltimore, MD, USA, 2018.
- [181] N. van Saberhagen. *CryptoNote v 2.0*. 2013. URL: <https://cryptonote.org/whitepaper.pdf> (visited on 02/25/2020).
- [182] Various Contributors. *Bitcoin: Core integration/staging tree*. github.com, 2020. URL: <https://github.com/bitcoin/bitcoin/network/members> (visited on 01/28/2020).
- [183] E. Wagner, A. Völker, F. Fuhrmann, R. Matzutt, and K. Wehrle. “Dispute Resolution for Smart Contract-based Two-Party Protocols”. In: *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2019, pp. 422–430.
- [184] G. Wang, Z. J. Shi, M. Nixon, and S. Han. “Sok: Sharding on blockchain”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019, pp. 41–61.
- [185] Wikipedia. *Zero Knowledge Contingent Payment*. 2018.
- [186] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger: EIP-150 REVISION*. 2016. URL: <https://gavwood.com/paper.pdf> (visited on 02/15/2020).

Bibliography

- [187] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. “Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation”. In: *Advances in Cryptology – CRYPTO 2019*. 2019, pp. 733–764.
- [188] A. C.-C. Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th Annual symposium on foundations of computer science*. Toronto, Canada, 1986, pp. 162–167.
- [189] M. Zamani, M. Movahedi, and M. Raykova. “Rapidchain: Scaling blockchain via full sharding”. In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada, 2018, pp. 931–948.
- [190] Zcash Foundation. *Privacy-protecting digital currency*. 2019. URL: <https://z.cash/> (visited on 02/15/2020).
- [191] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. “Town crier: An authenticated data feed for smart contracts”. In: *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016, pp. 270–282.
- [192] F. Zhang, P. Daian, I. Bentov, and A. Juels. “Paralysis Proofs: Safe Access-Structure Updates for Cryptocurrencies and More”. In: *IACR Cryptology ePrint Archive* (2018). URL: <http://eprint.iacr.org/2018/096>.