On the (im)possibility of building off-chain protocols from minimal assumptions

Vom Fachbereich Informatik der
Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

von

Siavash Riahi, M.Sc.
geboren in Tehran, Iran

Gutachter: Prof. Sebastian Faust
Associate Prof. Zekeriya Erkin
Dr. Julian Loss

Darmstadt 2023

Author: Siavash Riahi
Title: On the (im)possibility of building off-chain protocols from minimal assumptions
Ort: Darmstadt, Technische Universität Darmstadt

Datum der Einreichung: 10.02.2023
Datum der mündlichen Prüfung: 24.03.2023

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

Jahr der Veröffentlichung der Dissertation auf TUprints: 2023

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-243996
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/24399

Urheberrechtlich geschützt / In copyright
https://rightsstatements.org/page/InC/1.0/

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.

Darmstadt, den 10.02.2023

Siavash Riahi

Scientific Career

September 2012 till July 2016: Bachelor of Science in Computer Science Sharif University of Technology, Iran, Tehran

October 2016 till August 2018: Master of Science in Computer Science Saarland University, Germany

November 2018 till March 2023: Doctoral studies at Technical University of Darmstadt, Germany at the Chair of Applied Cryptography under the supervision of Prof. Sebastian Faust.

Acknowledgments

A Ph.D. is nothing like the other degrees that one can pursue at a university or school. It's a journey that takes years with many ups and downs. One not only requires courage to start this journey but also persistence, tenacity, and determination to see it through. Naturally, it is almost impossible to successfully finish a Ph.D. without the support of others. I would like to take this opportunity to express my gratitude toward everyone who helped me during the past years.

I would like to start with my advisor, Sebastian Faust, who not only believed in my capabilities but also provided constant feedback and support throughout all stages of my Ph.D. life. I am grateful for having the opportunity to visit multiple summer schools and conferences, during these past years. It was a pleasure being a member of the Chair of Applied Cryptography and having the opportunity to do my Ph.D. in this group.

Second, I would also like to thank the members of my defense committee, Julian Loss, Zekeriya Erkin, Reiner Hähnle, Marc Fischlin and Thomas Schneider for accepting to be in my defense committee and giving me the opportunity to defend my thesis in their presence.

I am very grateful to Jacqueline Wacker and Dorothee Nikolaus for helping me navigate the bureaucracy at TU-Darmstadt. Furthermore, I appreciate the support of the DFG and TU-Darmstadt for providing the funding and necessary equipment needed for my Ph.D. research. I would also like to thank CROSSING, especially Stefanie Kettler and Jacqueline Brendel, for organizing many research talks, workshops, and retreats to help us broaden our knowledge and find interesting research projects.

During my Ph.D. I spent many hours discussing interesting scientific problems with my colleagues and co-authors. I learned a lot from each and every one of them and would like to thank them for having many fruitful discussions and strenuous times before the deadline.

My deepest gratitude goes to my sister and parents who believed in me every step of the way, supported me both financially and emotionally throughout my whole life. You were always there for me when I needed you the most , and I am very grateful to have such a wonderful family.

List of Own Publications

Peer-reviewed Publications

[6] N. Alkeilani Alkadri, P. Das, A. Erwig, S. Faust, J. Krämer, S. Riahi, and P. Struck. "Deterministic Wallets in a Quantum World". In: *ACM CCS 2020*. Ed. by J. Ligatti, X. Ou, J. Katz, and G. Vigna. ACM Press, Nov. 2020, pp. 1017–1031. DOI: 10.1145/3372297.3423361.

[11] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi. "Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures". In: *ASIACRYPT 2021, Part II*. Ed. by M. Tibouchi and H. Wang. Vol. 13091. LNCS. Springer, Heidelberg, Dec. 2021, pp. 635–664. DOI: 10.1007/978-3-030-92075-3_22. Appendix A.

[12] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez. "Bitcoin-Compatible Virtual Channels". In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 901–918. DOI: 10.1109/SP40001.2021.00097. Appendix B.

[50] P. Das, A. Erwig, S. Faust, J. Loss, and S. Riahi. "The Exact Security of BIP32 Wallets". In: *ACM CCS 2021*. Ed. by G. Vigna and E. Shi. ACM Press, Nov. 2021, pp. 1020–1042. DOI: 10.1145/3460120.3484807.

[62] S. Dziembowski, G. Fabianski, S. Faust, and S. Riahi. "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma". In: *ITCS 2021*. Ed. by J. R. Lee. Vol. 185. LIPIcs, Jan. 2021, 72:1–72:20. DOI: 10.4230/LIPIcs.ITCS.2021.72. Appendix D.

[70] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi. "Two-Party Adaptor Signatures from Identification Schemes". In: *PKC 2021, Part I*. Ed. by J. Garay. Vol. 12710. LNCS. Springer, Heidelberg, May 2021, pp. 451–480. DOI: 10.1007/978-3-030-75245-3_17. Appendix C.

[73] A. Erwig, J. Hesse, M. Orlt, and S. Riahi. "Fuzzy Asymmetric Password-Authenticated Key Exchange". In: *ASIACRYPT 2020, Part II*. Ed. by S. Moriai and H. Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 761–784. DOI: 10.1007/978-3-030-64834-3_26.

[74] A. Erwig and S. Riahi. "Deterministic Wallets for Adaptor Signatures". In: *Computer Security – ESORICS 2022*. Ed. by V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng. Cham: Springer Nature Switzerland, 2022, pp. 487–506. ISBN: 978-3-031-17146-8.

Articles in Submission

[71] A. Erwig, S. Faust, and S. Riahi. *Large-Scale Non-Interactive Threshold Cryptosystems Through Anonymity*. Cryptology ePrint Archive, Report 2021/1290. https://eprint.iacr.org/2021/1290. 2021.

[72] A. Erwig, S. Faust, S. Riahi, and T. Stöckert. *CommiTEE: An Efficient and Secure Commit-Chain Protocol using TEEs*. Cryptology ePrint Archive, Report 2020/1486. https://eprint.iacr.org/2020/1486. 2020. Appendix E.

My Contribution

Many research projects require extensive collaboration between multiple co-authors. This makes specifying each person's contribution very difficult, if not impossible. The publications underlying this thesis are no exception. They are the result of many constructive discussions with my supervisor and co-authors. I thank all of them for their contribution to our publications. Parts of this thesis (e.g., some figures) are taken verbatim from these publications. Here, I aim to specify my contribution to these publications when possible and to the best of my knowledge.

Chapter 2 is based on [11, 12], joint works with Lukas Aumayr, Oğuzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, and Pedro Moreno-Sanchez. In [11], Andreas and I mainly worked on formalizing adaptor signatures as a standalone primitive, and proving that adaptor Schnorr and ECDSA signatures are secure in our model. We further worked on the security proofs of the generalized channel protocol. Generalized channel and its ideal functionality was proposed by Kristina and the application section is written by her and Oğuzhan. In [12], Andreas and I mainly worked on designing the virtual channel protocol without validity time and its security proofs. Oğuzhan and Lukas designed virtual channels with validity time. Kristina, Andreas, and I worked on modeling the security properties and ideal functionality of virtual channels. Lukas worked on the implementation and evaluation of both papers.

Chapter 3 is based on [70], jointly written with Andreas Erwig, Sebastian Faust, Kristina Hostáková, and Monosij Maitra. All authors had many discussions on the formalization of two-party adaptor signatures and the generic transformations. I designed the generic transformations from identification schemes to single and two-party adaptor signatures and two-party signatures. Afterwards, I (with Andreas and Kristina's help) proved the security of our generic single and two-party adaptor signature transformations. I also proved that building adaptor signatures from unique signatures is impossible. Andreas designed a formal model for two-party signatures and two-party adaptor signatures and wrote down the security proof of the generic two-party signature. Monosij and I formalized the properties that the underlying schemes must satisfy, to be compatible with our transformations, and I showed that Schnorr, Katz-Wang, and Guillou-Quisquater signatures satisfy them.

Chapter 4 is based on [62], jointly written with Stefan Dziembowski, Grzegorz

Fabiański, and Sebastian Faust. All authors had many rounds of discussions during different phases of the paper writing process. I put forward the main idea behind this work, namely, analyzing the data unavailability attack and its consequences on plasma MVP and Cash. I (with the help of the other authors) also worked on the model and security properties of plasma protocols. Finally, I provided the description and security proof of Plasma MVP and Cash. Grzegorz mainly focused on the lower bound proof which separates Plasma protocols into two distinct categories, i.e., fungible and non-fungible Plasma.

Chapter 5 is based on [72], jointly written with Andreas Erwig, Sebastian Faust and Tobias Stöckert. [72] is based on Tobias's M.Sc. thesis that Sebastian, Andreas, and I supervised. All authors discussed how the results of the thesis can be improved. I worked on the model and security properties as they had some flaws in the thesis. I also proposed a new protocol that significantly reduces the communication with the blockchain and argued that it is secure. Finally, I proposed two extensions to: (1) detect if the TEE has been compromised, and (2) support multiple Operators. Andreas also helped with writing down the new protocol and the extensions. Tobias mainly worked on the implementation/evaluation of this protocol.

Stefan, Sebastian, Matteo, Monosij, Pedro, (and Kristina in [70]) were the general advisor of the corresponding works and contributed with continuous feedback during all phases of the paper writing process.

Abstract

Blockchains have come a long way since the introduction of Bitcoin in 2008. Cryptocurrencies have become a household name as more people and even countries see the appeal in a secure decentralized ledger capable of processing monetary transactions and executing programs. Yet, one of the drawbacks of such decentralized systems is their lack of scalability. Hence, blockchains are unfortunately not ready to replace the existing financial system or cost-effectively execute programs. One class of solutions, proposed to tackle these limitations, are *off-chain* protocols. These protocols shift the communication away from the blockchain, by allowing parties to mostly communicate directly with each another. This direct communication is also referred to as off-chain communication. Probably the most well-known off-chain solution developed to date are Payment Channel Networks (PCNs). PCNs allow parties to make monetary transactions off-chain. Recently, more advanced off-chain solutions such as virtual channels, state channels and Plasma protocols have been developed for the Ethereum blockchain. These solutions allow making payments with improved efficiency and even executing programs (called smart contracts) off-chain. However, they rely on the fact that the Ethereum blockchain can execute Turing complete smart contracts, and it was unclear if one can build such protocols over more restricted blockchains such as Bitcoin.

In this thesis, we start by showing that virtual and state channels can be built over Bitcoin and similar blockchains. First, we present a new channel solution called *generalized channels* over Bitcoin. Generalized channels are comparable to state channels over Ethereum, i.e., generalized channels allow parties to execute applications off-chain that are supported by the underlying blockchain. In order to design generalized channels, we formalized a new primitive called adaptor signatures for the first time and show that Schnorr and ECDSA instantiations of this primitive are secure in our model. We then show that virtual channels can also be built over Bitcoin and Bitcoin-like blockchains. Virtual channels improve the efficiency of PCNs by reducing the communication needed for making off-chain payments. We further analyze the security of our protocols in the Universal Composability framework of Cannetti.

We continue by extending our adaptor signature formalization and model two-party adaptor signatures. This extension helps improve the efficiency of our

generalized channel construction. We provide two generic transformations that allow us to instantiate single and two-party adaptor signature schemes from signature schemes built from identification schemes that satisfy certain properties. We show that the Schnorr, Katz-Wang, and Guillou-Quisquater signature schemes satisfy the necessary properties required by our transformations and can generically be transformed into single and two-party adaptor signatures. Finally, we show that it is impossible to transform unique signatures schemes such as BLS into adaptor signature schemes.

After showing how to instantiate generalized and virtual channels over more restricted blockchains such as Bitcoin, we turn our attention to an alternative off-chain protocol called Plasma. In this solution, a single operator is responsible for updating parties' balances off-chain according to their transactions. On a high level, there are two classes of Plasma protocols, Plasma Cash and Plasma MVP, each with its advantages and disadvantages. Many in the Ethereum community focused on building a protocol that inherits the best properties of both classes without suffering from their disadvantages. We show that it is impossible to build a protocol that achieves the best of both worlds. Put differently, there is an inherent separation between Plasma Cash and MVP. This result can also be seen as "bringing order" to the huge landscape of Plasma protocols discussed in the Ethereum community. We further provide a formal model for Plasma protocols and also present instantiations of Plasma Cash and MVP that are secure in our model.

Finally, we conclude this thesis by presenting *CommiTEE*, an efficient yet simple Plasma protocol using a Trusted Execution Environment (TEE). A TEE is a piece of hardware that guarantees the correct execution of programs and secure storage of secret values. We only require the operator to have access to a TEE, and hence the end users are not burdened with purchasing expensive equipment. Our protocol removes many of the drawbacks seen in other Plasma constructions and offers a practical solution for real-world usage.

Zusammenfassung

Seit der Einführung von Bitcoin im Jahre 2008 hat sich um das Thema *blockchain* sehr viel getan. *Kryptowährung* wurde ein bekannter Begriff, insbesondere da mehr und mehr Menschen und sogar ganze Länder den Reiz eines sicheren dezentralen Verzeichnisses erkennen, das in der Lage ist finanzielle Transaktionen und komplexe Programme auszuführen. Allerdings leiden solche dezentralen Systeme nach wie vor an mangelnder Skalierbarkeit. Daher sind blockchains auch weiterhin nicht bereit, existierende finanzielle Systeme zu ersetzten oder eine kosteneffiziente Rechenplattform zu liefern. Eine Klasse an Lösungen, welche die Skalierbarkeitsproblem angehen, sind sogenannte *off-chain* Lösungen. Diese Protokolle reduzieren die Kommunikation mit der blockchain, indem sie es Parteien ermöglichen hauptsächlich direkt miteinander zu kommunizieren. Eine solche Kommunikation wird als off-chain Kommunikation bezeichnet. Die bisher wahrscheinlich bekannteste off-chain Lösungen sind Payment Channel Networks (PCNs). PCNs erlauben es Parteien finanzielle Transaktionen off-chain auszuführen. In den letzten Jahren wurden weitere und fortgeschrittenere off-chain Lösungen für die Ethereum blockchain entwickelt, wie beispielsweise Virtual Channels, State Channels und Plasma Protokolle. Diese Technologien ermöglichen effizientere off-chain Zahlungen und die off-chain Ausführung von komplexeren Programmen, sogenannten Smart Contracts. Allerdings verlassen sich diese Protokolle auf die Tatsache, dass die Ethereum blockchain Turing-vollständige Smart Contracts ausführen kann. Bisher war es unklar, ob solche Protokolle auch für eingeschränktere blockchains, sowie Bitcoin, entwickelt werden können.

Wir beginnen diese Thesis, indem wir zeigen, dass Virtual und State Channels auch auf Bitcoin und vergleichbaren blockchains aufgesetzte werden können. Zuerst präsentieren wir eine neue auf Bitcoin aufsetzende Channel Technologie, die wir Generalized Channels nennen. Generalized Channels sind mit State Channels auf Ethereum vergleichbar. Das bedeutet, sie erlauben die off-chain Ausführung jeglicher Programme, die auch von der darunter liegenden blockchain unterstützt werden. Als Baustein für unsere Generalized Channels führen wir eine neue Primitive ein, welche Adaptor Signaturen genannt wird. Wir formalisieren diese Primitive, zeigen dass sie basierend auf Schnorr und ECDSA instanziiert werden kann, und beweisen die Sicherheit der Instanziierungen in unserem Modell. Anschließend

zeigen wir, dass auch Virtual Channels auf Bitcoin und ähnlichen blockchains aufgesetzt werden können. Virtual Channels verbessern die Effizienz von PCNs, indem sie die für off-chain Zahlungen notwendige Kommunikation reduzieren. Wir analysieren und beweisen die Sicherheit unserer Protokolle in Canettis Universal Composability Framework.

Im nächsten Teil der Thesis erweitern wir die Formalisierung von Adaptor Signaturen und modellieren eine Variante der Primitive für zwei Parteien. Mit dieser Erweiterung verbessern wir die Effizienz unserer Generalized Channel Konstruktion. Anschließend stellen wir zwei generische Transformationen vor, welche es uns ermöglichen die Einparteien- beziehungsweise die Zweiparteienvariante der Primitive zu instanziieren. Ausgangspunkt der Transformationen sind Signaturverfahren, die auf Identifikationsverfahren basieren und bestimmte Eigenschaften erfüllen. Wir zeigen, dass Schnorr, Katz-Wand und Guilou-Quisquater Signaturverfahren die von unseren Transformationen benötigten Eigenschaften erfüllen und damit generisch in die Einparteien- und Zweiparteienvariante der Adaptor Signaturen transformiert werden können. Abschließend beweisen wir, dass es unmöglich ist eindeutige Signaturverfahren, wie beispielsweise BLS, in Adaptor Signaturverfahren zu transformieren.

Nachdem wir gezeigt haben, wie Generalized und Virtual Channels auf eingeschränkten blockchains wie Bitcoin aufgesetzt werden können, wenden wir unsere Aufmerksamkeit einem alternativen off-chain Protokoll namens Plasma zu. In Plasma ist eine einzige Partei, der Operator, dafür zuständig das Guthaben aller beteiligten Parteien off-chain basierend auf den Transaktionen im System zu updaten. Plasma Protokolle können grob in zwei Kategorien mit unterschiedlichen Vor- und Nachteilen eingeteilt werden, Plasma Cash und Plasma MVP. Die Ethereum Community investierte viel Aufwand darauf ein Protokoll zu entwickeln, welches die Vorteile beider Klassen kombiniert ohne unter den jeweiligen Nachteilen zu leiden. Wir zeigen, dass dies unmöglich ist, oder in anderen Worten, dass es eine inhärente Trennung zwischen Plasma Cash und MVP gibt. Dieses Ergebnis hilft die riesige Menge an unterschiedlichen Plasma Protokollen etwas weiter zu ordnen. Zusätzlich stellen wir ein formales Modell für Plasma Protokolle auf und präsentieren Instanziierungen von Plasma Cash und MVP, welche in unserem Modell sicher sind.

Wir schließen die Thesis mit der Vorstellung von *CommiTEE* ab, einem effizienten aber einfachen auf *Trusted Execution Environments* (TEE) basiertem Plasma Protokoll. Eine TEE ist ein Hardware Element, welches die korrekte Ausführung von darin laufenden Programmen und sichere Speicherung deren Werten garantiert. In unserem Protokoll muss nur der Operator Zugriff auf eine TEE haben. Es ist nicht notwendig, dass die Nutzer sich zusätzliche teure Ausrüstung zulegen.

Durch das beseitigen vieler der bisherigen Nachteile von Plasma Protokollen, bietet CommiTEE eine praktische Lösung für Echtweltanwendungen.

Contents

1. Introduction **1**
1.1. Off-Chain Solutions Overview 3
1.1.1. Off-Chain Channels 3
1.1.2. Plasma/Commit-Chain Protocols 7
1.2. Contribution and Thesis Outline 9

2. Bitcoin Compatible Generalized and Virtual Channels **12**
2.1. Preliminaries 13
2.2. Lightning-Style Channels 14
2.3. Our Contribution On Generalized Channels 17
2.3.1. Adaptor Signatures 18
2.3.2. Generalized Channels 20
2.4. Bitcoin Compatible Virtual Channels 21
2.4.1. Challenges on Designing Virtual Channels Over Bitcoin . . . 22
2.4.2. Overview of Our Solution 23
2.4.3. Our Concrete Instantiation of Virtual Channels 24
2.5. Related Work 27
2.5.1. Research on Payment Channels 27
2.5.2. Adaptor Signatures Related Work 29
2.6. Discussion and Future Work 30

3. Two Party Adaptor Signatures from Identification Schemes **32**
3.1. Preliminaries 32
3.2. Our Contribution 33
3.2.1. From $\mathsf{SIG}^{\mathsf{ID}}$ Schemes to $\mathsf{aSIG}^{\mathsf{ID,R}}$ 35
3.2.2. From $\mathsf{SIG}^{\mathsf{ID}}$ Schemes to $\mathsf{SIG}_2^{\mathsf{ID}}$ 38
3.2.3. From $\mathsf{SIG}_2^{\mathsf{ID}}$ Schemes to aSIG_2 39
3.3. Related Work 40
3.4. Discussion and Future Work 42

4. Lower Bounds for Plasma Protocols **43**
4.1. Preliminaries 44

4.2. Our Contribution . . . 46
4.2.1. Our Plasma Model . . . 46
4.2.2. Plasma Categories . . . 47
4.2.3. Separation result between Plasma MVP and Cash . . . 50
4.3. Related Work . . . 52
4.4. Discussion and Future Work . . . 53

5. CommitTee 55
5.1. Our Contribution . . . 55
5.1.1. Background on TEEs . . . 55
5.1.2. Security and Efficiency Properties . . . 56
5.1.3. CommiTee Protocol . . . 56
5.1.4. Evaluation of CommiTee . . . 59
5.2. Related Work . . . 59
5.3. Discussion and Future Work . . . 61

6. Conclusion 62

7. Bibliography 66

Appendix A. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures 83

Appendix B. Bitcoin-Compatible Virtual Channels 114

Appendix C. Two-Party Adaptor Signatures From Identification Schemes 150

Appendix D. Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma 181

Appendix E. CommiTee: An Efficient and Secure Commit-Chain Protocol using TEEs 204

1. Introduction

The introduction of Bitcoin by Satoshi Nakamoto in 2008 [132] initiated a new branch of financial products called *Cryptocurrencies*. Bitcoin put forth an innovative approach, called *blockchain*, to instantiate a public ledger where parties can make monetary transactions in a secure and decentralized way. One of the main goals of Bitcoin is to make monetary transfers more accessible by removing financial institutes as the middlemen who charge astronomical fees and can even refuse to process certain transactions. Instead, Bitcoin relies on a large set of parties operating in an open peer-to-peer network to maintain a public ledger. The role of these maintainers, often called *miners*, is to gather users' transactions, verify their authenticity (using digital signatures) and validity according to a clear-cut set of rules. These transactions are then accumulated in a new block of fixed size and proposed to the network. As the name suggests, a blockchain is nothing but a list of blocks that are "chained" together via a cryptographic process executed by the miners. This process creates an immutable, append-only, public list of blocks that can be used to emulate the role of a financial institute such as a bank. To compensate these miners for their effort, they receive some fees directly from the users who are making transactions.

Bitcoin's implementation was released to the public in 2009 when the financial markets were suffering from the 2008 housing crisis. This made Bitcoin emerge as a reliable alternative, compared to the failing financial institutes. Although at the beginning it only gained the attraction of a few, Bitcoin quickly became very well-known. After its success, many new blockchains were built and deployed, examples of which can be found in [22, 92, 178]. According to one estimate [119], there are more than 10000 tradable cryptocurrencies in circulation today. The most prominent blockchain after Bitcoin is Ethereum [178]. While Bitcoin aims to be the "world's decentralized bank", Ethereum's goal is to be the "world's decentralized computer". More precisely, Ethereum allows parties to execute Turing complete programs called *Smart Contracts* over the blockchain. This makes Ethereum quite suitable for executing programs in a decentralized and secure way. Naturally, parties need to compensate the miners for executing the programs by paying them fees according to the complexity of the program being executed.

The main difference between Bitcoin and Ethereum is in their transaction systems.

Bitcoin uses the Unspent Transaction Output (UTXO) model. Intuitively, in the UTXO model money is transferred from an unspent transaction to a new unspent transaction. On a high-level the transactions in this model form a directed acyclic graph. This transfer of money can also be conditioned on some event, e.g., the new transaction must be digitally signed by a certain party. The spending conditions of a UTXO transaction are also called scripts. Ethereum uses an account based system where each user and smart contract has a balance. Parties can specify in their transaction how much money they wish to send to another party or which function of a smart contract they wish to call. Ethereum has a Turing complete scripting language, and can execute arbitrary programs. By scripting language, we are referring to the set of program instructions that the blockchain can process. One of the main reason for using the UTXO transaction system is its simplicity and security. There are countless examples where flawed smart contract code causing users to lose their money on Ethereum. The most famous of these attacks is the DAO hack [162]. The simplicity of UTXOs does not allow for the execution of complex programs which in return results in much less critical bugs.

One might ask: Can we employ blockchains to process daily monetary transactions or to execute complex programs? To answer this question, we first have to understand how the blockchain is maintained in an open peer-to-peer network.

Virtually all blockchains use a consensus mechanism to guarantee that new blocks are generated correctly and only contain valid transactions. There are many approaches to reach consensus between the miners in a peer-to-peer network. However, the most prominent method, which is used in Bitcoin, and also up until recently in Ethereum, is Proof of Work (PoW) [58, 93]. In PoW, to propose a new block (of a fixed pre-determined size), a miner needs to solve a time- and resource-consuming cryptographic puzzle. This results in Bitcoin and Ethereum blockchains not being able to compete in terms of average transaction throughput with other centralized payment systems. As an example, Bitcoin can on average only process less than 7 transactions per-second [28, 29] but the Visa payment system can process around 24.000 transactions per second [2].

There have been many proposals to improve the scalability of blockchains. One line of work, also called *layer-1* solutions, focuses on improving blockchains' consensus mechanism. One example of such a proposal is Proof of Stake [16, 41, 47, 53, 105]. Another layer-1 scalability proposal is to split the set of total transactions submitted to the network into smaller chunks called *shards*. Each shard is then processed by a subset of the miners which can increase blockchain's transaction throughput [5, 108, 111, 116, 176, 180]. Implementing layer-1 solutions however, requires changing the consensus mechanism of the underlying blockchain. Yet in a decentralized cryptocurrency, some miners might simply disagree with the

proposed changes and continue using the old procedures. This would split the cryptocurrency into two (incompatible) cryptocurrencies, one following the new consensus mechanism and another one following the old one. As an example, some members of the Bitcoin community proposed increasing the size of each block. Yet, many miners disagreed with this change. This split the currency into two parallel blockchains. Those miners who agreed with this change created Bitcoin Cash [27].

As changing the consensus mechanism of a blockchain is a hard and lengthy process, a new line of solutions, called *off-chain protocols* or *layer-2* solutions, emerged. The main idea here is to shift the communication away from the blockchain and instead allow parties to mostly communicate privately. We denote the communication made with the blockchain by *on-chain* and the private communication between parties without involving the blockchain is called *off-chain*. This approach not only reduces the amount of data that the underlying blockchain needs to process and reach consensus on, but also reduces the amount of fees parties have to pay, which makes blockchains more accessible. The main difficulty of designing off-chain protocols is achieving similar security guarantees compared to when parties communicate directly with the blockchain.

Due to the importance of off-chain protocols, our main research questions in this thesis are:

1. *Can we instantiate off-chain solutions deployed over more capable blockchains, i.e., Ethereum, on more restricted blockchains such as Bitcoin?*

2. *Do some of the more recent (and publicized) off-chain proposals suffer from inherent limitations?*

Let us first briefly summarize the most prominent off-chain solutions.

1.1. Off-Chain Solutions Overview

We divide off-chain solutions into two main categories: (1) Channels, and (2) Plasma (Commit-Chain) protocols. Let us give a brief overview of each category.

1.1.1. Off-Chain Channels

One can view a payment channel as a direct line of credit between two parties. The parties can update the coin distribution in this channel many times and eventually report the final result to the blockchain. We first start by explaining the concept of (simple) payment channels and then show how this concept can be extended into more complex payment mechanisms.

Simple Payment Channels. The first and by far most widely studied category of off-chain solutions is payment channels [54, 89, 152, 163]. In a nutshell, parties create a channel by "blocking" their coins on-chain. They can afterwards update the distribution of coins in this channel off-chain. Let us elaborate more via an example. We will build on this example throughout this section to motivate the solutions and point out their drawbacks. Assume Alice buys a cup of coffee every day and wishes to pay the shopkeeper Bob via a blockchain. As she makes recurring payments to Bob, she can avoid making an on-chain transaction every day by opening a payment channel with Bob and locking, say 10$ worth of coins, in this channel. Every day, Alice and Bob update the state of the channel and reduce Alice's, and increase Bob's balance accordingly. Eventually, they can close their channels via an on-chain transaction and make the final payout.

Let us now summarize the main advantages of this solution: (a) Alice and Bob only have to make 2 on-chain transactions (one for opening and one for closing the channel) regardless of how many times they wish to make off-chain payments, and (b) the payment is instantaneous, i.e., upon the state being updated, both parties know that they can get the payout on-chain. Yet in this solution, Alice's locked coins can only be used for paying Bob. In other words, if she wishes to visit multiple coffee shops, she needs to open multiple channels and lock coins in all these channels. Payment Channel Networks (PCNs) were introduced in order to overcome this drawback.

Payment Channel Networks. It is clear that opening a new channel between all pairs of parties who wish to make a payment is financially not feasible. Yet, the parties who have already opened a channel with each other form a network or graph. The vertices of this graph represent the parties and the edges are the channels between these parties. The idea of a Payment Channel Network is to allow routing transactions between parties in this graph. By doing so, it would no longer be necessary to open a new channel between each pair of parties who wish to make off-chain transactions. Alice in this scenario can make payments to any other coffee shop as long as she can find a path in the graph and route her transaction. This improves the "pure" channel solution as it does not require parties to build multiple channels to pay multiple parties off-chain. As a consequence, parties do not need to lock coins multiple times for each payment channel. Examples of PCNs are the Lightning network [152] over Bitcoin and the Raiden network over Ethereum [155]. Nevertheless, this solution also has its own drawbacks. To elaborate more, assume Alice wishes to pay Bob but needs to route her transaction through Charlie and Dave as shown in Fig. 1.1. Although Alice has 10 coins in her channel with Charlie, she can at most pay 3 coins to Bob. This is because

Dave only has 3 coins in his channel with Bob. In general, routing a transaction in a PCN is far from trivial. Even worse, some intermediaries might simply refuse to route transactions. Furthermore, the intermediaries do not simply route transactions out of the kindness of their hearts, they usually request some fees from the sender/receiver. Although lower than the on-chain fees, this cost factor needs to be considered when routing a transaction. One natural consequence of these

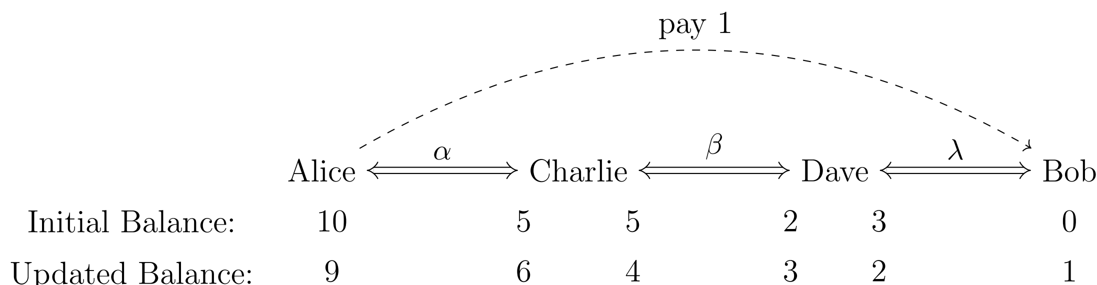

Figure 1.1.: A PCN with three ledger channels α, β and λ. In this example, Alice sends 1 coin to Bob.

drawbacks is the emergence of a few *hubs* with whom most parties are connected. Payment Channels Hubs (PCHs) [60, 85, 90, 169], allow parties to simply open a channel with a hub and rout all transactions via this hub. Although having a centralized party to make payments is undesirable, PCHs almost completely circumvent the issues caused via complex routing algorithms in PCNs. Overall PCNs and PCHs have two common drawbacks: (a) the intermediaries need to be online for each transaction they route, and (b) the intermediaries need to lock quite a huge amount of coins in each channel in order to facilitate transactions. The money that an intermediary needs to lock in order to route payments is also called *collateral*. The second drawback is somewhat inherent to the fundamental idea and approach behind payment channels. Hence, a completely new off-chain solution (i.e., Plasma or Commit-Chain which we will explain later) was developed to overcome it. However, the first drawback can be solved via a concept called *virtual channels* which we will explain next.

Virtual Channels. The main idea of a virtual channel can be summarized as an off-chain channel built over other off-chain channels. Let us continue with our example. Consider a scenario where Alice and Bob both have a channel with Ingrid. A virtual channel is a direct channel between Alice and Bob that does not require Ingrid to route transactions and update the state of the channel. More precisely, Alice, Ingrid, and Bob first run a protocol to open a "virtual" channel between Alice

and Bob (see Fig. 1.2 for a pictorial illustration of a virtual channel). Afterwards, Ingrid no longer needs to route transactions and Alice and Bob can update the state of this channel independently. Eventually, Alice and Bob close their virtual channel and their balances are updated accordingly in their channels with Ingrid. The idea behind virtual channels was first introduced in [60]. However, the solution in [60] requires the underlying blockchain to be able to execute Turing complete smart contracts. An extension of payment and virtual channels allow parties to execute smart contracts off-chain. This type of channels are called *state channels*.

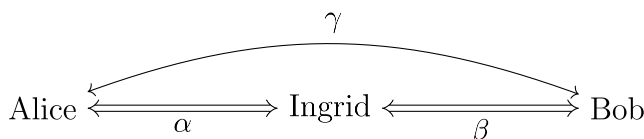

Figure 1.2.: A virtual channel γ built over ledger channels α, β.

State Channels. The concept of state channels was first introduced for channels executed over the Ethereum blockchain. State channels allow parties to not only make payments off-chain, but also execute smart contracts. As an example, if Bob wishes to determine the price of Alice's coffee based on the type and size of the order, he can use a smart contract to charge Alice accordingly. By using a state channel they can execute this smart contract off-chain. Using state channels would save parties huge amounts of fees as they do not need to submit any transactions on-chain in order to execute the smart contract. State and virtual state channels were first formalized in [63]. In [59], the authors extended the previous work to build multi-party (virtual) state channels, i.e., channels where more than two parties can make payments or execute smart contracts. However, virtual channels and state channels introduced in [59, 60, 63] require the underlying blockchain to support executing Turing complete programs. It was unknown (until our work) if one can build virtual or state channels for blockchains with more limited capabilities such as Bitcoin.

Security Proofs

To guarantee that off-chain protocols are secure and parties do not lose their coins, one must exhibit a security proof. This proof must guarantee that an honest party does not lose his or her coins even if all other parties are malicious and collude with each other. The golden standard for proving the security of off-chain protocols, especially channel solutions, is the Universally Composability (UC)

framework [43, 44]. In UC, one must prove that the execution of the protocol is indistinguishable from the ideal behavior that is expected from the protocol, even if some of the parties behave maliciously. The ideal behavior expected from the protocol is defined via a so-called *ideal functionality*. More precisely, it must be impossible for the malicious parties to force an outcome that is not captured in the ideal functionality. Another approach for proving the security of cryptographic primitives and protocols is the game-based approach. Here, one or more *bad* events are defined in an experiment (also called game). The security proof in this case involves showing that these bad events cannot happen i.e., the adversary cannot win the games. While game based proofs are mainly used for primitives such as digital signatures, UC is used to prove the security of complex protocols. Protocols proven secure in the UC framework can be easily composed with each other and the composed protocol would still remain secure.

Channel Solutions Summary. To summarize, off-chain channels can reduce the cost of making payments and even executing applications for users. However, we identify the following two drawbacks: (a) some of the more advanced solutions such as virtual and state channels are only available on Ethereum, as they rely on complex operations that are not supported on more restricted blockchains such as Bitcoin, and (b) the total amount of collateral needed to facilitate payments via PCNs and PCHs is rather high.

1.1.2. Plasma/Commit-Chain Protocols

Plasma (later called Commit-Chain in the literature) was first proposed by Poon and Buterin [151] as an alternative off-chain solution. The main goal of this solution is to reduce the amount of collateral needed for a hub when routing transactions. Let us first calculate the amount of collateral needed in the PCH solution for a simple scenario. Assume that Alice is connected to a hub called Ingrid and wishes to make transactions freely and without any restrictions with all other parties connected to Ingrid. If there are in total n parties connected to this hub, where party i has a balance of c_i in his or her channel with Ingrid, Ingrid must have $\sum_{i\in\{1,\cdots,n\}\setminus i_a} c_i$ coins in her channel with Alice alone (i_a is Alice's index in the set of n parties). Therefore, in practice, no hub has enough coins locked to facilitate arbitrary transactions between parties. Hence, at some point the hub's channels would be depleted. When Ingrid's balance in Alice's channel is depleted, Alice can no longer receive any coins. She must either first spend some coins such that Ingrid's balance increases or close this channel on-chain and open a new channel with Ingrid. Alternatively she can find a new path in the PCN. However, none of these solutions are ideal.

On a conceptual level, one can allot this high collateral requirement to the fact that payment channels were originally designed for two parties only. More precisely, parties can only transfer the coins that they indeed have locked in the channel to each other. As PCHs are just an extension of normal payment channels, the intermediary must have enough balance in all her channels to facilitate the transfer of coins. Yet, the role of an intermediary is mainly to "move money around", similar to what we have in normal financial systems. Poon and Buterin [151] proposed a compromise, namely to sacrifice the instant finality of the channels in order to remove/reduce the amount of collateral. Their main idea is that the intermediary, who is called the *operator* in this setting, gathers the transactions of all the users connected to it and periodically updates the balances of the parties. Naturally, one requires a complex mechanism to prevent a malicious operator from stalling the system by going offline or incorrectly updating the balances of the parties.

Upon explaining their initial idea, the Ethereum community came up with proposals on how to build Plasma protocols [38, 40, 102, 129, 131, 142, 144, 174]. However, the community failed to construct an efficient solution inheriting the best properties of all proposals. To make matters worse, most of the proposals were simply discussed over the Ethereum research forum and lacked formalism [76]. Therefore, analyzing them became a tedious task.

Rollups. After the introduction of Plasma protocols, the community moved to a new proposal called *Rollups*, first proposed by Barry Whitehat [177]. This solution can be seen as a combination of layer-1 and layer-2 solutions as it is not a fully off-chain solution. In Rollups, the operator gathers the transactions and posts all of them on-chain, in the most compact form possible, e.g., the signatures for these transactions are not posted on-chain. However, the blockchain does not verify these transactions, it only stores them for a short period of time to make sure that all parties can see the list of transactions sent to the operator. This makes sure that the operator cannot keep some parties in the dark. There are two general classes of Rollups, (a) Optimistic-Rollups, and (b) ZK-Rollups. In Optimistic-Rollups, the operator periodically publishes the raw transaction data. Hence, the operator can post invalid transactions, e.g., by allowing parties to spend more coins than they actually own, or posting a transaction that was never signed by the sender. To deal with a malicious operator, the parties need to challenge the operator on-chain in case they detect any misbehavior. Examples of Optimistic-Rollups are Arbitrum and Optimism [98, 139]. ZK-Rollups, first introduced in [39, 177], take a different approach and use a cryptographic primitive called Succinct Non-Interactive Argument of Knowledge (SNARK). For simplicity, the community sometimes calls this primitive zero-knowledge proof. A SNARK enables a prover to

convince a verifier that a statement is true in a succinct way. Ideally, the amount of data that the prover needs to send to the verifier is independent of the statement being proven. Such proof systems can even show that a certain program was executed correctly on some input [23]. In ZK-Rollups, the operator is prevented from behaving maliciously by providing such a proof. In a nutshell, the operator generates a proof to convince the blockchain that the raw data posted on the blockchain only consists of valid transactions. Naturally, if the operator is honest, optimistic-Rollups are more efficient than ZK-Rollups since the operator does not need to compute a SNARK when posting the information on-chain. However, if the operator cheats, parties need to make on-chain transactions to challenge him or her. This can be undesirable since parties have to pay fees to make on-chain transactions. Hence, choosing between ZK- and optimistic-Rollups depends on the use case and how trustworthy the operator is. We note that compared to Plasma protocols, Rollups are simpler to design as the raw transaction information is indeed posted on-chain. In Plasma protocols, this is not the case and the operator posts the transactions made by the parties only off-chain. This makes designing secure Plasma protocols much more difficult as the operator can simply stop sending the list of transactions to the users.

1.2. Contribution and Thesis Outline

Our goal in this thesis is to analyze both channel and Plasma/Commit-Chain solutions, explore the minimal requirements for building them, and at the same time expose their limitations. The organization of this thesis is as follows:

Chapter 2 As mentioned before, it was not clear if off-chain solutions such as state and virtual channels can be built over blockchains such as Bitcoin that do not support smart contracts and can only execute limited operations called scripts. In Chapter 2, we present the works, [11] and [12]. First, we introduce *Generalized Channels*, an extension to normal payment channels that support off-chain execution of scripts that are supporter by the underlying blockchain. Hence, state channels can be seen as a special case of generalized channels that are only designed for blockchains with a Turing complete scripting language. Our generalized channel construction uses a new primitive called *Adaptor Signatures*. We provide the first standalone formalization of adaptor signatures and prove that the single-party constructions of adaptor signatures from Schnorr [158] and ECDSA [94] are secure in our model. Our solution is also much more efficient than the state-of-the-art payment channel solution over Bitcoin, namely the Lightning channel [152], both in terms of off-chain and on-chain communication complexity.

After successfully showing how to build generalized channels, in [12] we show how to use them in order to build (generalized) virtual channels on blockchains that only support a limited scripting language, such as Bitcoin. We note that building virtual channels over Bitcoin was an open problem as previous solutions used Turing complete smart contracts in order to build virtual channels. We further evaluate our virtual channel protocol and compare it with the existing PCN solutions. For n sequential payments, our solution only requires exchanging $9 + 2n$ transactions off-chain, which results in $3524 + 695n$ bytes. Routing a payment in a PCN requires exchanging $8n$ transactions which results in a total of $3026n$ bytes. Hence, after the first payment, our solution would become much more efficient than the existing PCN protocols.

We model both protocols from [11, 12] in the UC framework and argue the security of our constructions in this model.

Chapter 3 As mentioned before, we use adaptor signatures to instantiate generalized and virtual channels over Bitcoin. However, in [11] we only formalized the single-party adaptor signatures. Yet as we will see later in Chapter 3, one could improve the efficiency of our generalized channels protocol if two parties could jointly generate an adaptor signature. Furthermore, it was unclear until our work if signature schemes other than ECDSA and Schnorr can be transformed into adaptor signatures. In Chapter 3, we present the paper [70]. We provide the first formalization of two party adaptor signature schemes and show how to generically build single and two-party adaptor signatures, via (almost) generic transformations from signature schemes built from identification schemes [106] (which we denote by $\mathsf{SIG}^{\mathsf{ID}}$). We further prove that it is impossible to build adaptor signature schemes from unique signatures. Unlike [106], our transformations are not fully generic, i.e., the underlying $\mathsf{SIG}^{\mathsf{ID}}$ schemes must satisfy certain properties to be compatible with our transformations. Nevertheless, we show that Schnorr, Katz-Wang, and Guillou-Quisquater signatures [87, 101, 159] satisfy these properties.

Chapter 4 In this chapter, we focus on Plasma protocols and present our paper [62]. In this work, we "bring order" to the huge landscape of Plasma protocols and analyze their limitations. We provide a model for Plasma protocols and categorize them into fungible Plasma also called Plasma MVP [38], and non-fungible Plasma also called Plasma Cash [40]. We also show how Plasma MVP and Cash protocols can be instantiated and proven secure in our model. In non-fungible Plasma protocols, parties own and exchange unique tokens. This means a party who wishes to leave the system must withdraw each token separately on-chain which increases the total communication cost needed with the blockchain. On the other hand in fungible Plasma parties have a single balance and can withdraw all their balance

with one on-chain transaction. Yet, this class of protocols requires a more complex design to prevent a malicious operator from stealing parties' coins. In Plasma MVP, if the operator is malicious, honest parties have to challenge him or her on-chain in time to prevent the operator from stealing their coins.

In the Ethereum community, many have tried to design the best of both protocols. We show that there is an inherent separation between these classes of protocols. In other words, to prevent a malicious operator from stealing parties' coins, either: (a) all honest parties have to communicate a lot with the blockchain, even though they did not intend to, or (b) when parties wish to exit their coins and use them on-chain, they need to communicate a lot with the blockchain. The first case happens in Plasma MVP where honest parties have to challenge a malicious operator. The second case happens in Plasma Cash where each token has to be withdrawn separately. Put differently, our results shows that one cannot build a Plasma protocol that has the best properties of Plasma MVP and Cash and does not suffer from their disadvantages.

Chapter 5 In the final work presented in this thesis [72], we design an efficient Plasma protocol called CommiTEE by using a Trusted Execution Environment, or TEE for short. A TEE is a chip (usually embedded in a CPU) that guarantees correct execution of the programs that it runs. We only assume that the operator has access to the TEE while the end users who wish to make off-chain transactions do not need to have one. By using a TEE, we build a fungible Plasma protocol that is much simpler than the existing proposals. Furthermore, our protocol requires minimal interaction with the blockchain, which drastically reduces the cost of executing our protocol. We also show that our solution can be extended to allow (a) changing the operator, i.e., if the main operator behaves maliciously or crashes, and (b) detecting if the TEE was compromised. Our experimental implementation over Ethereum shows that CommiTEE is at least 2 times (and in some cases more than 16 times) cheaper in terms of communication complexity compared to existing Plasma implementations. Furthermore, compared to NOCUST-ZKP [102], which uses zero-knowledge proofs (instead of TEEs) to guarantee the correct execution of programs, CommiTEE decreases the on-chain costs by a factor of ≈ 19.

2. Bitcoin Compatible Generalized and Virtual Channels

As already mentioned in the introduction, blockchains do not scale. This is mainly due to the decentralized nature of blockchains and the fact that executing a consensus algorithm among many parties in a peer-to-peer network is time and resource consuming. This makes blockchains less likely to be widely adopted. Off-chain protocols add a new layer on top of the existing blockchain (i.e., *layer-1*) to achieve better scalability. Although there are many different off-chain solutions such as [20, 86, 95, 176], payment channels [30, 55, 152] are the most prominent ones. The life cycle of payment channels can be divided into three phases. First, parties make an on-chain transaction to "lock" their coins and *open a channel*. Afterwards, they can *make transactions* off-chain by updating the distribution of coins in their channel. The *state* of a channel determines how the coins should be distributed between the parties. Parties usually need to exchange digital signatures on this new state to indicate that they agree with the update. Finally, parties *close the channel* by posting the latest state of the channel on-chain. Hence, a payment channel allows two parties to update the coin distribution of the channel without requiring parties to make on-chain transactions. The main challenge of designing payment channels is making sure that parties can only close their channel in its latest state and according to the final coin distribution.

Recall that the Ethereum blockchain can execute Turing Complete programs. This makes building applications such as payment, state and virtual channels [59, 63, 126] much easier over Ethereum, compared to a more restricted blockchain such as Bitcoin. For example, on Ethereum a smart contract can store multiple values, or compare integers and make payments to parties according to its logic. This is quite useful to compare different states of a channel and identify the newer state using a version number. Hence, if a malicious party tries to close the channel by posting an outdated state, the honest party can submit the latest state and the blockchain can determine which state is more recent by comparing the version number of these states. This approach is not possible over Bitcoin. To understand Bitcoin's restrictions, we first give some background and explain how transactions on Bitcoin look like. Furthermore, we show how Lightning payment channels, the

most widely used payment channels over Bitcoin, are built.

2.1. Preliminaries

Digital Signatures. A digital signature scheme allows a party to validate the authenticity of a message. More precisely, a digital signature scheme is a triple of algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$. $\mathsf{Gen}(1^n)$ is a Probabilistic Polynomial Time (PPT) algorithm that on input the security parameter 1^n outputs a secret and public key pair (sk, pk). On a high-level, the security parameter roughly indicates how long the keys have to be. $\mathsf{Sign}_{sk}(m)$ is a PPT algorithm that on input a secret key sk and a message $m \in \{0,1\}^*$, outputs a signature σ. Finally, $\mathsf{Vrfy}_{pk}(m, \sigma)$ is a Deterministic Polynomial Time (DPT) algorithm that on input a public key pk, a message m and a signature σ, outputs a bit b. We say that a signature scheme is secure or *Unforgable* if no PPT attacker, called *adversary*, can forge a valid signature on a fresh message (for which it has not seen any signatures before).

Hash Functions. On a high level, a hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^l$ is a function that compresses its input. However, it should be difficult for a PPT algorithm to find two inputs that $\mathcal{H}$ maps to the same value. More precisely, no PPT adversary can generate two (non-equal) values x_1 and x_2 such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$.

For a more detailed definition and explanation on digital signatures and hash functions we refer the reader to [100].

UTXO Transaction system. Bitcoin's transaction system follows the Unspent Transaction Output (UTXO) model. We use figures to show how transactions are processed and used in our applications. As an example, Fig. 2.1 illustrates two UTXO transactions. The transaction tx on the left-hand side is published on the blockchain. This is illustrated by the double lined rounded frame. The transaction tx' on the right is not yet published on the blockchain and therefore has only a single line rounded frame. Each transaction has a set of inputs and outputs. In this model, coins are held in *outputs* of a transaction. In Fig. 2.1, tx has two outputs one with the value x_1 and the other with the value x_2. tx' on the other hand has only one output (we have omitted the inputs in these figures for simplicity). More precisely, an output θ is a tuple consisting of two elements (cash, φ), where cash is the amount of coins associated to this output and φ specifies the spending condition of this output. These conditions (also called scripts) must be satisfied by the transaction that wants to spend this output. In other words, the spending transaction must provide a *witness*, $\mathsf{Witness}$, that satisfies the spending condition in order to be able to spend this output. More formally, a transaction tx is a tuple of the form $(\mathsf{txid}, \mathsf{In}, \mathsf{Out}, \mathsf{Witness})$. Let us go through each item one by one. txid is

the unique identifier assigned to tx. It is generally computed by hashing the input and output of tx, i.e., $\mathsf{txid} := \mathcal{H}(\mathsf{In}, \mathsf{Out})$. In is the list of tx's inputs. Note that these inputs are in fact outputs of other transactions. Hence, each element in In identifies the transaction via its unique id and the index of the output. Out is a list of outputs associated to tx, i.e., $\mathsf{Out} := [\theta_1, \cdots, \theta_n]$. $\mathsf{Witness}$ is the witness that allows tx to spend the inputs it has listed.

Let us now briefly talk about the spending conditions of the outputs of tx and tx'. If the spending transaction must be signed w.r.t. a specific list of public keys, say $pk_1, \cdots, pk_n$, this list is mentioned below the arrow coming out of the output, e.g., the first output of tx must be signed by A w.r.t. pk_A. We say that this output "belongs" to A, or A is the owner of this output. The rest of the conditions are mentioned above the arrow. The two main conditions that we are interested in are hash-locks and time-lock. Hash-locks are shown by simply mentioning the value h on top of the arrow and time-locks by $+t$ over the arrow exiting an output. An output with hash-lock value h can only be spent by a transaction that provides x, where x is the preimage of h, i.e., $\mathcal{H}(x) = h$. An output with time-lock value t can only be spent after it was published on the blockchain for more than t rounds. We use rounds as a unit of time in this work. A round is roughly equal to a second and is the amount of time needed for a party to send a message to another party. Finally, if an output can be spent under different spending conditions we use the gray diamond symbol and multiple arrows exiting the same output, as depicted in Fig. 2.1 for tx'.

Figure 2.1.: Two example transactions demonstrating our notations.

2.2. Lightning-Style Channels

Although there are many proposals on how to construct a payment channel on Bitcoin, the most prominent and widely used channel network to date is the Lightning network [152]. The main challenge of designing payment channels is making sure that parties can only close their channel in its latest state and according to the final coin distribution. Bitcoin's scripting language does not allow comparing multiple values. Hence, the approach used to build channels on Ethereum does not

work over Bitcoin. Lightning style channels use a punishment mechanism to protect against users who try to close the channel in an outdated state. In Lightning channels, upon updating the coin distribution, parties exchange a secret value which allows the other party to "punish" him or her in case this outdated state is posted on the blockchain. We will now explain in more details how Lightning channels work with the help of a pictorial illustration in Fig. 2.2. This figure illustrates the transactions that can end up on the blockchain in a Lightning channel.

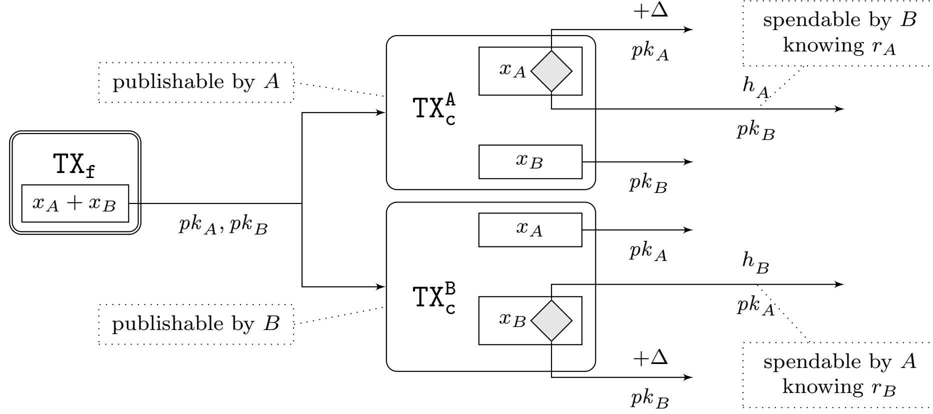

Figure 2.2.: A Lightning style payment channel where Alice and Bob have x_A and x_B coins respectively. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ is the upper bound for the time needed to publish a transaction on a blockchain.

Opening the channel: When opening a channel, Alice and Bob need to post a so called "funding transaction", $\mathtt{TX_f}$, on-chain. $\mathtt{TX_f}$ is funded by both parties, each contributing x_A and x_B coins respectively to this channel. $\mathtt{TX_f}$ can be spent by a transaction signed by both parties. Before signing and publishing $\mathtt{TX_f}$, both parties agree on the transactions that can spend $\mathtt{TX_f}$. Alice and Bob exchange signatures on two transactions $\mathtt{TX_c^A}$ and $\mathtt{TX_c^B}$ which are called the "commit transactions". These transactions specify the coin distribution of the channel. More specifically, Bob signs $\mathtt{TX_c^A}$ and sends it to Alice and Alice signs $\mathtt{TX_c^B}$ and sends it to Bob. Alice sings $\mathtt{TX_c^A}$ locally and is the only party who can post it on-chain. Bob respectively signs $\mathtt{TX_c^B}$ locally. Let us first see how $\mathtt{TX_c^A}$ distributes the $x_A + x_B$ coins. Upon Alice publishing $\mathtt{TX_c^A}$, Bob can directly spend his x_B coins. However, Alice can only spend x_A coins after Δ rounds. Bob can also spend this output if he has the preimage of the hash value h_A, i.e., he knows r_A such that $h_A = \mathcal{H}(r_A)$. We will later elaborate more on why Bob can spend this output knowing r_A. $\mathtt{TX_c^B}$ distributes

the coins analogously. Note that when generating the commit transactions, Alice and Bob choose r_A and r_B at random (without revealing them to the other party) to calculate h_A and h_B used in the commit transactions.

Updating the channel: Alice and Bob first sign and exchange two new commit transactions (with updated coin distribution and new hash values calculated as $h'_A = \mathcal{H}(r'_A)$ and $h'_B = \mathcal{H}(r'_B)$), similar to the process used during the creation of the channel. Afterwards they exchange the pre-images of the hash values of the old commit transactions. More specifically, Alice sends r_A to Bob such that $h_A = \mathcal{H}(r_A)$ and Bob sends r_B to Alice where $h_B = \mathcal{H}(r_B)$. The process of exchanging these pre-images are also called *revocation* of the commit transaction.

Closing the channel: To close the channel, either party can publish his or her commit transaction.

Punishment: Let us elaborate more why Bob can spend the output with x_A coins in $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ (see Fig. 2.2). When updating this channel, parties sign and exchange new commit transactions. They furthermore invalidate the old state by exchanging the pre-images of h_A and h_B. Now if Alice posts a revoked commit transaction, say $\mathtt{TX}_\mathtt{c}^\mathtt{A}$, to the blockchain, the parties must have exchanged the "revocation" secrets for this transaction. In this case, r_A must have been sent to Bob. Using r_A, Bob can spend the x_A coins immediately. By doing so he gets *all* the coins in this channel and hence he does not lose money regardless of the coin distribution in the most recent state of the channel. Alice is punished for publishing a revoked $\mathtt{TX}_\mathtt{c}^\mathtt{A}$. Note that Δ must be long enough to give Bob time to react and punish Alice. Put differently, Alice should only be able to spend the x_A coins after Bob had enough time to punish her.

Lightning channels have two major drawbacks. Consider channels that support executing multiple applications. As an example, assume Alice and Bob wish to open new channels on top of the current channel, i.e., the output of the commit transaction would fund a new funding transaction. This application might seem a bit artificial, but virtual channels, which we will explain in Sec. 2.4, are an example of channels being built on top of other channels. In this case, the commit transaction would have multiple outputs for different applications that are being executed off-chain, each with its own spending condition. For this use case, Lightning-style channels suffer from the following drawbacks:

Revocation per Output. If we naïvely extend these channels to allow for the execution of multiple applications, each application would require a separate punishment mechanism. Hence, parties have to store a revocation secret for each application. In case an outdated commit transaction is posted, each application

has to be punished separately, i.e., we have revocation per-application and not per-channel. This increases the total communication with the blockchain.

Two Commit Transactions. Although having two commit transactions is not an issue for simple payments, building applications on top of Lightning channels might result in less efficient solutions. To realize the application mentioned above, parties need to sign two funding transactions as they do not know in advance which of the commit transactions would end up on-chain. Hence, to update the new channels built on top the old channel, they need to repeat the update process two times, once for each funding transaction.

Following the above discussion we asked ourselves the following questions:

1. *Can we design a channel protocol for Bitcoin with only one commit transaction and a single revocation mechanism per channel?*
2. *Is it possible to instantiate virtual channels over blockchains with limited scripting language such as Bitcoin?*

We answer both questions positively. First, we introduce *Generalized Channels* in Sec. 2.3. These channels can execute (2-party) applications (that are supported by the underlying blockchain) off-chain. Our instantiation of generalized channels only has a single commit transaction and revocation mechanism per-channel. Afterwards, in Sec. 2.4 we show how to build virtual channels on blockchains with a restricted scripting language. Our work has been disseminated in the following two articles and can be found in Appendix A and B:

[11] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi. "Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures". In: *ASIACRYPT 2021, Part II.* Ed. by M. Tibouchi and H. Wang. Vol. 13091. LNCS. Springer, Heidelberg, Dec. 2021, pp. 635–664. DOI: 10.1007/978-3-030-92075-3_22. Appendix A.

[12] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez. "Bitcoin-Compatible Virtual Channels". In: *2021 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, May 2021, pp. 901–918. DOI: 10.1109/SP40001.2021.00097. Appendix B.

2.3. Our Contribution On Generalized Channels

In our parer [11], we answer the first question mentioned above. Let us first give an intuition why we need two commit transactions in Lightning style channels.

By having two commit transactions, it is easy to identify which party started the closing procedure of the channel and can be potentially punished. Note that only Alice can publish $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ since Bob does not receive her signature on this transaction. Respectively, only Bob can publish $\mathtt{TX}_\mathtt{c}^\mathtt{B}$. Hence, to build a channel with only one commit transaction, we need a mechanism to identify the party who has published this commit transaction.

Although this might look impossible at first sight, we can achieve it using *Adaptor Signatures*. We formalized this primitive for the first time and showed that adaptor signature constructions from Schnorr [147, 148], and the single party variant of the ECDSA proposal [141] (with some slight modifications), are indeed secure in our model. We first give an overview on adaptor signatures in Sec. 2.3.1 and then present the solution overview for our generalized channels in Sec. 2.3.2.

2.3.1. Adaptor Signatures

An adaptor signature is a primitive that ties together the authorization of a message and the leakage of a secret value. Generally speaking, this primitive is used in protocols with two parties, say Alice and Bob, where Alice has the signing secret key sk_A and Bob has an instance of a hard relation, i.e., $(y, Y) \in \mathsf{R}$. Informally, R is a hard relation if just given Y, it is difficult for a PPT algorithm to compute y. Bob would like to have a signature on a message m under Alice's secret key. Alice on the other hand would like to make sure that if her signature is published by Bob, she will learn Bob's secret, i.e., y. Adaptor signatures allow Alice and Bob to achieve this functionality. Given the secret key sk_A and the statement Y, Alice can generate a *pre-signature* on the message m. A pre-signature is an incomplete signature that would not verify under the signer's public key. Bob can complete this pre-signature using the witness y into a full signature. Alice can extract y given the full signature that Bob generated and the pre-signature. Naturally, upon receiving the pre-signature, Bob wants to be sure that he can indeed complete (or adapt) it into a full signature using y. Alice on the other hand wants a guarantee that Bob can only generate a signature on the message m. Furthermore, Bob should not be able to generate this signature in a malicious way such that she cannot extract y.

More precisely, an adaptor signature is defined w.r.t. a hard relation R and a signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ and consists of four algorithms $\Xi_{\mathsf{R},\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ where:

$\mathsf{pSign}_{sk}(m, Y)$: outputs a pre-signature $\tilde{\sigma}$ on input a secret key sk, message m, and statement Y.

$\mathsf{pVrfy}_{pk}(m, Y, \tilde{\sigma})$**:** on input a public key pk, message m, statement Y, and pre-signature $\tilde{\sigma}$, outputs 1 if the pre-signature is valid and 0 otherwise.

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

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

We say that an adaptor signature scheme is *correct*, if given a pre-signature $\tilde{\sigma}$ computed as $\tilde{\sigma} = \mathsf{pSign}_{sk}(m, Y)$, a full signature σ generated by adapting $\tilde{\sigma}$, i.e., $\sigma = \mathsf{Adapt}(\tilde{\sigma}, y)$, and $(Y, y) \in \mathsf{R}$, the following conditions hold: (1) $\tilde{\sigma}$ is valid, i.e., $1 = \mathsf{pVrfy}_{pk}(m, Y, \tilde{\sigma})$, (2) σ is valid, i.e., $1 = \mathsf{Vrfy}_{pk}(m, \sigma)$, and (3) given $\tilde{\sigma}$ and σ, it is possible to extract a correct witness y' for the statement Y, i.e., $(Y, \mathsf{Ext}(\sigma, \tilde{\sigma}, Y)) \in \mathsf{R}$.

An adaptor signature must satisfy the following 3 security properties.

Existential Unforgeability This property is similar to the unforgeability of normal signature schemes. It guarantees to the holder of the secret key that no PPT adversary is able to forge a valid signature under this user's public key on a fresh message m, i.e., a message for which no signature was previously returned to the adversary. However, in case of adaptor signatures, the adversary receives an additional pre-signature on the message for which it wishes to forge a signature.

Pre-signature Adaptability This property guarantees to the receiver of a pre-signature that regardless of how a pre-signature is produced, if it is valid, it can be completed into a valid full signature. Put differently, an adversary cannot generate a pre-signature maliciously such that pVrfy returns 1, but it cannot be completed into a valid full signature. More precisely, consider a pre-signature $\tilde{\sigma}$ which is valid, i.e., $\mathsf{pVrfy}_{pk}(m, Y, \tilde{\sigma}) = 1$. Given the corresponding y (i.e., $(Y, y) \in R$), the full signature generated as $\sigma = \mathsf{Adapt}(\tilde{\sigma}, y)$ must be valid as well, i.e., it must hold that $\mathsf{Vrfy}_{pk}(m, \mathsf{Adapt}(\tilde{\sigma}, y)) = 1$.

Witness Extractability This property states that given a valid pre-signature $\tilde{\sigma}$ generated with respect to a message and statement pair (m, Y) of the adversary's choice, he or she cannot forge a signature σ such that the corresponding y cannot be extracted from the signature, pre-signature pair $(\tilde{\sigma}, \sigma)$. This property guarantees to the party who is generating the pre-signature that if a (malicious) receiver publishes a valid full signature, he or she can indeed extract the witness.

We refer the reader to our full paper [11], in Chapter A, for the formal model, Schnorr and ECDSA adaptor signatures constructions and proving that they are secure in our model. We will explore adaptor signatures more in Chapter 3.

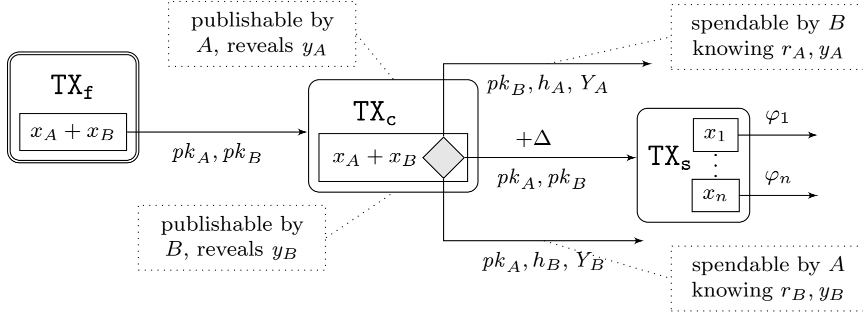

Figure 2.3.: A generalized channel with outputs $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain.

2.3.2. Generalized Channels

We now use adaptor signatures to build channels with a single commit transaction and revocation per-channel. We call these kind of channels *Generalized Channels* as they can support efficient off-chain execution of arbitrary applications that the underlying blockchain supports. Let us explain how generalized channels work by going through the transaction flow depicted in Fig. 2.3.

Our main idea is to force parties to reveal a secret when posting the commit transaction. This secret value can be used by the other party to punish him or her if necessary. More precisely, to punish Alice, Bob needs to know two secret values (1) the revocation secret r_A, and (2) the witness y_A (see the outputs of $\mathtt{TX_c}$ in Fig. 2.3). Knowing r_A shows this state is revoked and knowing y_A guarantees that Alice was the party who posted the outdated commit transaction. r_A is exchanged similar to the Lighting-style channels during the update procedure of the channel. On the other hand, y_A is revealed by using adaptor signatures. Concretely, Alice and Bob exchange pre-signatures on the single commit-transaction $\mathtt{TX_c}$ with respect to their statements Y_A and Y_B. If Alice wishes to post $\mathtt{TX_c}$, she needs to first adapt Bob's pre-signature $\tilde{\sigma}_B$ on $\mathtt{TX_c}$, to a full signature σ_B using y_A. Therefore, when publishing $\mathtt{TX_c}$ Bob can extract y_A as he knows both $\tilde{\sigma}_B$ and σ_B. Hence, unlike Lightning channels, we do not need two commit transactions to identify the party who posted the commit transaction and can potentially be punished.

We furthermore, decouple the applications and the punishment mechanism by introducing a new *split transaction* $\mathtt{TX_s}$ which contains all the outputs needed for the applications being executed. This transaction can spend the output of $\mathtt{TX_c}$

after Δ rounds if no party has punished the other party. Therefore, generalized channels have a single punishment mechanism per-channel and not per-application.

Let us now go through the creation, update, closing and punishment procedures of a generalized channel. To create a generalized channel, Alice and Bob first generate a statement witness pair, i.e., $(y_A, Y_A) \in \mathsf{R}$ and $(y_B, Y_B) \in \mathsf{R}$. They further randomly sample the revocation secrets and compute its hash value, i.e., (r_A, h_A) and (r_B, h_B) where $h_A := \mathcal{H}(r_A)$ and $h_B := \mathcal{H}(r_B)$ and exchange h_A, Y_A and h_B, Y_B. They then generate the $\mathtt{TX_f}$, $\mathtt{TX_c}$ and $\mathtt{TX_s}$ transactions using these exchanged values according to the spending condition described before and depicted in Fig. 2.3. Just like Lightning channels, before signing and posting the funding transaction, Alice and Bob exchange signatures on $\mathtt{TX_s}$ and *pre-signatures* on $\mathtt{TX_c}$. Finally, they sign and post $\mathtt{TX_f}$ on-chain.

To update the channel, parties only need to agree on new $\mathtt{TX_s}$ and $\mathtt{TX_c}$ transactions and exchange signature/pre-signature on them similar to the process of creating the channel. They further need to exchange their revocation secrets r_A and r_B similar to Lightning channels. Now if say Alice wishes to close the channel, she needs to complete Bob's pre-signature using y_A. Upon publishing $\mathtt{TX_c}$, Bob can extract her secret y_A as he has now both a full signature and pre-signature on $\mathtt{TX_c}$. If the state is already revoked, i.e., Alice and Bob have exchanged r_A and r_B, Bob can punish Alice by spending the output of $\mathtt{TX_c}$ unilaterally. We now summarize the security properties that generalized channels satisfy.

(S1) Consensus on creation: Both parties have to agree to create a channel.

(S2) Consensus on update: Both parties in the channel must agree to update the state of this channel.

(S3) Instant finality with punish: The latest state of the channel can be reflected back on the blockchain or if a party misbehaves and prevents the channel from being closed in the correct state, this party is punished, i.e., the honest party gets all the coins in this channel.

Our formal model, detailed protocol description, security proofs, and performance evaluation can be found in the full version of our paper [11] (Appendix A).

2.4. Bitcoin Compatible Virtual Channels

In [12], we show for the first time how to build virtual channels over Bitcoin. A Virtual channel allows two parties who both have a channel with a common

intermediary, to make direct off-chain payments to each other without requiring the intermediary to route the payments. We start with the challenges of designing virtual channels over Bitcoin in Sec. 2.4.1. We then continue with an overview of our solution in Sec. 2.4.2 and finally present our concrete instantiation in Sec. 2.4.3.

2.4.1. Challenges on Designing Virtual Channels Over Bitcoin

We now explain our idea on how to build virtual channels on a blockchain that uses UTXO transaction system with a limited scripting language. To build a virtual channel Alice, Bob and Ingrid update their channels, such that a new channel between Alice and Bob can be created. In other words, the coins in the channels between Alice, Ingrid and Bob, Ingrid, fund a new funding transaction for a channel between Alice and Bob. Naturally, this new funding transaction is not posted on-chain and is only kept off-chain between the parties.

There are two main challenges that we need to address when designing virtual channels over Bitcoin. First, we need to ensure that the opening and closing procedures of a virtual channel are atomic, i.e., either the state of both channels is updated or none of them. This is necessary to guarantee that Ingrid does not lose any coins. As an example, if Alice has received 10 coins from Bob in the virtual channel, after closing the virtual channel, Ingrid's balance in her channel with Alice decreases by 10 coins, i.e., Ingrid pays Alice 10 coins, but her balance in the channel with Bob increases by the same amount. Clearly if this process is not atomic, Ingrid can lose coins. Second, we need a punishment mechanism that guarantees either (1) the virtual channel is closed in its latest state, or (2) the honest parties are financially compensated. Let us now discuss how we address these challenges. In this section, we use the term *ledger channels* to refer to "normal" channels that have a funding transaction posted on-chain.

Synchronous Updates of the Ledger Channels: To guarantee that the underlying ledger channels are updated atomically, Ingrid must be the final party that signs off on the creation procedure. In other words, Ingrid must receive the update request from both Alice and Bob, and then decide if she signs them or not.

New Punishment Mechanism for Virtual Channels: Recall that in contrast to ledger channels, the funding transaction of a virtual channel is not posted on-chain. There are two possible scenarios in case of a dispute. First, if both ledger channels can be closed in their correct state, Alice and Bob can post the funding transaction of the virtual channel and transform it into a ledger channel. In this case, Alice and Bob can simply rely on the punishment mechanism of the underlying ledger channel. Second, if one of the channels, say the channel between Ingrid and Bob,

is maliciously closed in a state such that the funding transaction of the virtual channel cannot be posted on-chain anymore, Alice must be able to punish Ingrid and get financially compensated. Put differently, we need an additional punishment mechanism in case the funding transaction of the virtual channel cannot be posted on-chain.

2.4.2. Overview of Our Solution

First, we provide a modular framework for instantiating a virtual channel. The main idea is to connect the split transactions of the two ledger channels such that in case both channels are closed the funding transaction of the virtual channels can be posted on the blockchain. In other words, the outputs of these split transactions are the inputs of the virtual channel's funding transaction.

Creating a Virtual Channel. To create a virtual channel, both channels between Alice, Ingrid and Bob, Ingrid must be updated synchronously. In a nutshell, this is done in the following steps:

1. Alice and Bob indicate to Ingrid that they wish to update the channels and open a virtual channel. Ingrid can decide if she wishes to facilitate the virtual channel.
2. Alice, Bob and Ingrid generate the necessary transactions for the virtual channel. This process is called *virtual channel setup* and depends on the exact virtual channel construction.
3. Alice and Bob sign the new update transactions (in case of generalized channels, the commit and split transactions) and send them to Ingrid.
4. Finally, Ingrid signs the update transactions and sends them to both parties.

As we can see, Ingrid has the final say in the creation mechanism. This guarantees Ingrid that Alice and Bob cannot create asynchrony between the channels.

Updating a Virtual Channel. This process is quite similar to updating a ledger channel. The main difference arises in case one of the parties misbehaves and the channel needs to be closed on-chain. In this case, parties have to first close their underlying ledger channels, transform the virtual channel to a ledger channel and then continue the execution on-chain or punish one another.

Closing a Virtual Channel. Closing a virtual channel is quite similar to the creation procedure. The only difference here is that step 2, i.e., setting up the virtual channel is no longer needed. Instead, parties simply reflect their final balance in the virtual channel back to their ledger channel.

Offloading a Virtual Channel. Offloading a virtual channel means that the virtual channel is transformed into a ledger channel. This might be necessary in case Ingrid wishes to unlock her collateral or if parties are in dispute and want to punish each other on-chain. To offload a channel, the underlying ledger channels need to be closed and the funding transaction of the virtual channel posted on-chain.

Punishment in a Virtual Channel. There are two kinds of punishment in a virtual channel (1) punishment for not being able to offload the virtual channel, and (2) ledger channel punishment executed after offloading the virtual channel. The latter punishment mechanism is exactly the same punishment mechanism as in the underlying ledger channel, i.e., generalized channel's punishment mechanism. The former punishment mechanism on the other hand punishes the party who misbehaved and prevented the virtual channel from being offloaded. This punishment mechanism depends on the concrete instantiation of the virtual channel.

2.4.3. Our Concrete Instantiation of Virtual Channels

Now that we have described the challenges in designing a virtual channel over Bitcoin and described the different procedures of a virtual channel in our modular framework, let us go through our virtual channel construction.

Overview of the transaction flow. The full transaction flow for our virtual channel construction is in Fig. 2.4. In our description, we are assuming that both parties Alice and Bob together have a total of c coins in the virtual channel, i.e., $c = c_A + c_B$. We further assume that Ingrid receives f coins as fee for facilitating the creation of the virtual channel (here, each party pays $f/2$ coins to Ingrid but this fraction can be adjusted arbitrarily in practice). Let us go through the transaction flow step by step. First, recall that upon closing the underlying ledger channels in the correct state, it must hold that the funding transaction of the virtual channel can be posted on-chain. Hence, the outputs of both $\mathtt{TX}_\mathtt{s}^\mathtt{A}$ and $\mathtt{TX}_\mathtt{s}^\mathtt{B}$ are the input of $\mathtt{TX}_\mathtt{f}$ which is the funding transaction of the virtual channel. Upon $\mathtt{TX}_\mathtt{f}$ being posted on-chain Ingrid can get $c + f$ coins and Alice and Bob can now treat $\mathtt{TX}_\mathtt{f}$ as the funding transaction of a "normal" ledger channel. Now let us assume that one of the split transactions, e.g., $\mathtt{TX}_\mathtt{s}^\mathtt{B}$, is not posted on-chain before time Δ. In this case, Alice can get all the coins in the output of $\mathtt{TX}_\mathtt{s}^\mathtt{A}$ and punish Ingrid. Let us analyze offloaded and punishment cases separately.

Transaction flow in case of offloading. The transaction flow when the virtual channel is offloaded can be seen in Fig. 2.5. In this case, the funding transaction of the virtual channel is posted on-chain. This allows Ingrid to receive $c + f$ coins.

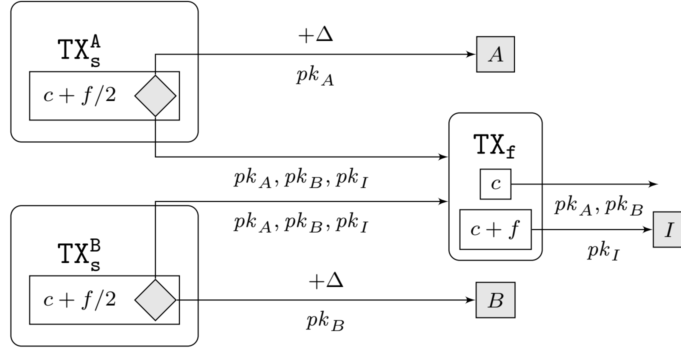

Figure 2.4.: Funding of a virtual channel γ without validity.

The rest of the coins can be used to publish the commit and split transactions of the virtual channel, which is now transformed into a ledger channel.

Transaction flow in case of punishment. Now assume the funding transaction cannot be posted on-chain because $\mathtt{TX}_\mathtt{s}^\mathtt{B}$ is not posted on-chain as depicted in Fig. 2.5. In this case, Alice can receive all the coins in her channel with Ingrid as she is no longer able to post the funding transaction of the virtual channel. By doing so Alice punishes Ingrid for not publishing $\mathtt{TX}_\mathtt{s}^\mathtt{B}$. In other words, Ingrid is responsible for making sure that her channel with Bob is not updated to a new state such that the virtual channel can no longer be offloaded. Note that if Bob closes his channel with Ingrid in a revoked state, she can punish him and receive $c + f/2$ coins. Hence, in this case only the malicious Bob is punished.

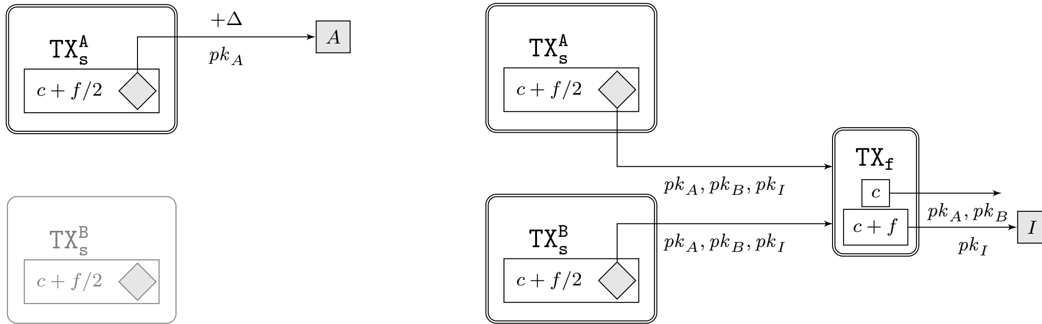

Figure 2.5.: The figure on the right shows the transactions published after the virtual channel is successfully offloaded. The figure on the left shows the transactions published after Alice successfully punishes Ingrid. The grayed transaction $\mathtt{TX}_\mathtt{s}^\mathtt{B}$ has not been posted on-chain.

The virtual channel protocol presented above can remain open for an unlimited

amount of time. Hence, it is important that Ingrid is also able to offload the virtual channel by closing the underlying ledger channels and release her collateral. If Ingrid was not able to offload this channel, Alice and Bob could collude and force Ingrid's collateral to be locked in the virtual channel indefinitely. We say that this type of virtual channel is *without validity*. In [12], we also present a virtual channel construction *with validity* where Alice and Bob can only use the virtual channel for a limited amount of time and must close their channel before the validity time. We refer the reader to our paper [12] for the formal model, detailed protocol description, security proofs, and our second instantiation of virtual channels with validity time. Here, we only give a high-level description of the security properties that our construction achieves.

Security and Efficiency properties of our construction

Let γ be a virtual channel as described above. Naturally, all the security properties that the ledger channels achieve, i.e., (S1), (S2) and (S3), hold for γ. In addition, γ satisfies the following security properties for the intermediary Ingrid:

(V1) Balance security: If Ingrid is honest, she never loses coins, even if Alice and Bob collude.

(V2) Offload with punish: Ingrid can transform a γ to a ledger channel. Alice or Bob can initiate the transformation as well. This transformation is either completed or the honest parties are financially compensated.

Our virtual channel construction is also efficient and does not require any on-chain transactions as long all parties are honest. More concretely, we have:

(E1) Constant round creation: If the creation of γ succeeds, the number of rounds needed to create this channel must have been constant.

(E2) Optimistic update: In the optimistic case, i.e., when both parties Alice and Bob are honest, updating γ takes a constant number of rounds.

(E3) Optimistic closure: In the optimistic case, i.e., when Alice, Bob and Ingrid are honest, closing γ takes only constant number of rounds.

Building Virtual Channels over Generalized vs Lightning-Style Channels

Our virtual channel can be built over Lightning-style channels as well as generalized channels. In fact, we intentionally designed our channels in a modular way in order

to allow their deployment over the Lightning network. However, as explained before, Lightning channels have two commit transactions which would make building virtual channels on top of them even less efficient. Let us briefly explain why. Each of the two Lightning ledger channels have two commit transactions (see Fig. 2.2). In order to guarantee that the $\mathtt{TX_f}$ of the virtual channel can be posted on-chain, parties need to consider all four possible combinations of commit transactions being posted on-chain. This means that four $\mathtt{TX_f}$ transactions need to be created for the virtual channel and therefore updating the state of the virtual channel also needs repeating the update procedure four times. This is a huge overhead. Our detailed performance evaluation can be found in our paper [12] (Appendix B).

2.5. Related Work

We divide the related work into two categories, payment channels and adaptor signatures presented in Sec. 2.5.1 and Sec. 2.5.2 respectively.

2.5.1. Research on Payment Channels

We now provide an overview of the related work for our payment channel protocols. An extensive list of such solutions can be found in works such as [95] and [86].

Uni-directional payment channels, where one party can pay and the other party can only receive coins, was first proposed by Spilman and Hearn [89, 163]. Decker and Wattenhofer [54] later extended this type of channel to allow both parties to send and receive coins. In 2016, Poon and Dryja [152] showed how to build a PCN called the Lightning network. However, at the time of proposal, Bitcoin's scripting language was much more limited than today and as such Lightning channels could not be deployed immediately over Bitcoin. In 2017, a new Bitcoin Improvement Proposal (BIP141) [67] was implemented by the Bitcoin community which finally allowed the Lightning channels and network to be executed over Bitcoin. At the time of writing Lightning network has more than 17000 nodes and 86000 channels which is quite impressive [1]. Lightning network was also formally modeled and analyzed in the UC framework by Kiayias and Litos [103].

After Lightning's success, a few alternative channel solutions were introduced for the Ethereum blockchain such as Sprites [127], counterfactual/state channels [165], Raiden Network [155] and Perun (virtual) payment and state channels [59, 60, 63].

Other than the general research done on designing new and innovating payment and state channel solutions, there have been some proposals on how to achieve specific features in channels and channel networks such as anonymous payments [56,

90, 120, 121, 169] or efficient and more secure multi-hop payments in a PCN [8, 13, 15, 65, 68, 97, 121]. However, we will not go into details on these works here and instead mainly focus on virtual channels related work.

Virtual Channels. The concept of virtual channels was first introduced in [60]. The authors proposed a solution for the Ethereum blockchain and discussed how their solution can be used to improve the efficiency of PCH solutions as the hub is no longer needed to route every single transaction. Afterwards, two followup papers extended this work. In [63], the concept of state channels and virtual state channels were formalized and the authors showed how to instantiate them over the Ethereum blockchain. In [59], the idea of virtual state channels was extended to multi-party (virtual) state channels over Ethereum. As the name suggests, a multi-party channel is a channel that is executed between more than two parties. In this case, updating a channel is a more complex task as all parties need to agree on the new state and be able to receive their coins on-chain. Concurrent to our work [96] also proposed virtual channels over Bitcoin. However, their construction only allows updating the state of the virtual channel for an a priori fixed number of times. This is mainly because the authors use a different technique to guarantee that only the latest state can be posted on-chain. Let us go a bit more into details. When opening the channels, the parties agree on an initial state which can only be posted on the blockchain after time t. This can be achieved using a different type of time-lock (called absolute time-lock) that specifies at what time this transaction is valid, and can be posted on-chain. During each update, parties exchange a new state with a smaller time-lock (e.g., $t' = t - \Delta$). This guarantees that the latest state can be posted on-chain before the older ones. However, at some point the time lock t of a new state would be equal to the current time and the channel can no longer be updated. Hence, parties either need to frequently close and open new virtual channels, or they must choose a rather large t. Frequently opening and closing virtual channels is counter-intuitive as virtual channels are designed to minimize the interaction with the intermediary. Choosing a large t can however force the parties' funds to be locked for a long time. Other than the restriction mentioned above, the construction in [96] is not a "generalized virtual channel", i.e., it does not support executing applications off-chain and only allow making payments.

After showing that it is possible to build virtual channels over Bitcoin [12], two followup works showed how to build "longer" and "recursive" virtual channels [14, 104]. In a bit more detail, in [14] the authors design a virtual channel over multiple intermediaries for Bitcoin. On the other hand, Kiayias et al., [104] show how to build virtual channels over virtual channels, i.e., Alice and Bob do not need a ledger

channel with Ingrid, they only need to have a virtual channel with Ingrid and can open a virtual channel on top of the existing virtual channels.

2.5.2. Adaptor Signatures Related Work

The notion of adaptor signatures was first mentioned by Poelstra [147, 148] for Schnorr signatures as an alternative to hash-locks. Given its utility, adaptor signatures slowly started to gain more popularity. Moreno-Sanchez and Kate [141] proposed a threshold two party construction for adaptor signatures based on ECDSA signature. In [121] the authors used adaptor signatures to design a PCN protocol with a more secure routing procedure. However, they did not provide a standalone definition for adaptor signatures that can be then used in other works such as ours. Concurrent to our work, Fournier [78] attempted to formalize adaptor signatures which he called one-time verifiable encrypted signatures. However, his definition lacked formalism and therefore he was not able to prove that the existing adaptor signature proposals were indeed secure in his model.

A primitive similar but weaker than adaptor signatures is *lockable signatures* [172]. Lockable signatures allow generating a locked signature lk which can later be unlocked into a full signature. However, unlike adaptor signatures, to generate the locked signature (which is equivalent to the pre-signature in adaptor signatures) the full signature must be produced in advance and the locked signature must be generated honestly. More precisely the locked signature is simply computed as $lk := \sigma \oplus \mathcal{H}(\sigma')$ where both σ and σ' are signatures. Put differently the signature σ is being locked by another signature σ'. When σ' is revealed, one can open the lock as $\sigma = lk \oplus \mathcal{H}(\sigma')$. Naturally, to calculate lk both σ and σ' must be computed in advance. Furthermore, just given, lk one cannot verify that it indeed locks a valid signature σ. Therefore, one needs more advanced cryptographic techniques such as secure multi party computation [42, 48] in order to guarantee that lk was generated honestly. On the positive side, lockable signatures can be built from any signature scheme. This is mainly because calculating lk only requires xoring the full signature with a hash value. Boneh et al., [33] defined a primitive called verifiably encrypted signatures. In their setting, the signer's goal is to prove that he or she has provided a ciphertext encrypting a valid signature on a message m. Banasik et al., [19] introduced a protocol that enables two parties, a buyer and a seller, to exchange a digital asset even if the underlying blockchain cannot execute Turing complete programs. However, these works do not provide a construction that can be used to instantiate adaptor signatures.

Impact of our work. Shortly after our work was published, [75, 170] proposed adaptor signature schemes that are post-quantum secure, i.e., these schemes remain

secure even if the adversary has a quantum computer while the users still use normal computers. In [75], Esgin et al., achieved post-quantum security by designing a scheme based on well-studied lattice assumptions. The authors of [170] build their scheme based on isogenies which are less analyzed. Both works further show how to build PCNs using their adaptor signature construction. In another work, Tairi et al., [169] showed how to build PCHs that achieve anonymity, i.e., the hub cannot link the identities of the sender and receiver. They later updated their paper and used the adaptor signature notation introduced in our work.

2.6. Discussion and Future Work

In this chapter, we presented two of our papers, namely generalized channels and virtual channels over Bitcoin [11, 12]. Our generalized channels have two main advantages over other Lightning channels, (1) generalized channels only require a single commit transaction per-channel while Lightning channels require 2 commit transactions, and (2) generalized channels have a single punishment mechanism per-channel while Lightning channels require a punishment mechanism per-application built on top of the channel. These two features, alongside the novel design of generalized channels, allow for the execution of applications supported by the underlying blockchains even if it only supports a limited scripting language. Hence, generalized channels can be seen as a generalization of state channels over Ethereum. Put differently, state channels are a special case of generalized channels that can only be deployed over blockchains with a Turing complete scripting language. To build generalized channels, we utilized adaptor signatures, a novel primitive that we formalized as a standalone primitive for the first time. We further proved that the Schnorr and ECDSA adaptor signature constructions are secure in our model.

Afterwards, we showed how to instantiate virtual channels over blockchains with limited scripting language such as Bitcoin. All previous constructions of virtual channels could only be deployed on blockchains such as Ethereum which support Turing complete smart contracts. Therefore, our works help improve the scalability and usability of more limited blockchains such as Bitcoin.

We modeled the concept of generalized and virtual channels in the Universal Composability (UC) framework [43, 44] and analyzed our protocols' security in this model. UC is the gold standard when it comes to modeling and analyzing the security of off-chain protocols. Protocols proven secure in this model can be executed in multiple parallel sessions and composed with other protocols secure in this model, with minimal overhead for proving that the composition is secure.

Since the publication of our papers, many articles have either used or introduced improvements to our work. As already mentioned before, works such as [14] and [104] present virtual channel solutions with extended functionality. Another example is [128] by Mirzaei et al., who proposed an extension to generalized channels by adding fair and privacy preserving watchtower support. Users of the channel can delegate the execution of the punishment mechanism to the watchtowers and hence do not need to monitor the blockchain constantly for misbehavior. This allows them to go offline if needed without the risk of losing funds. Mirzaei et al., were able to build their scheme because we detached the punishment procedure from the applications in generalized channels.

One main open direction for future work is modeling and designing multi-party generalized and virtual channels over blockchains with limited scripting language. As our generalized and virtual channels are only executed between two parties, they only support two party applications. Multi-party variants of our constructions will be able to execute multi-party applications supported by the underlying blockchain. We will later see how to build 2-party adaptor signatures in Chapter 3, however an interesting question is whether multi-party generalized and virtual channels require multi-party adaptor signatures or is it possible to instantiate them using the single party adaptor signatures presented in this section.

Another more general direction of future work is introducing a new modeling framework, other than UC, for off-chain and blockchain applications. Although UC can be used to model such protocols, it is a general purpose model. Blockchain protocols have many common elements that can be embedded in a tailor-made model in order to reduce the overhead of modeling protocols and proving their security. As an example, all such protocols communicate with a common ledger, i.e., blockchain. There are many ways to model such a ledger in UC, some are easier to use when proving the security of the protocols such as the ones used in [11, 12, 59, 60, 63], and others are more realistic yet much more difficult to use when analyzing the security of protocols [17, 103, 104]. Overall, coming up with a new model that simplifies the modeling and proof process of blockchain related protocols would streamline the security analysis and provide the protocol designers with a universal language to describe their protocols.

A similar future work direction is modeling protocols in a formal/automated verification tool. There are many verification tools such as ProVerif [31], Tamarin [157] or EasyCrypt [64]. However, there is still a huge gap between the capabilities, and ease of usability of these tools and the necessary requirements for analyzing complex protocols. Therefore, extending such tools and improving their usability for complex blockchain protocols would be an interesting future work direction. An example of a tool that tries to reduce this gap is Verifpal [107].

3. Two Party Adaptor Signatures from Identification Schemes

We have already seen that adaptor signatures are quite useful for building off-chain solutions such as generalized and virtual channels [11, 12]. However, adaptor signatures were only constructed from Schnorr and ECDSA signatures, and it was unclear if they can be built from other signature schemes as well. Furthermore, only single-party adaptor signature schemes were formalized in these works. Yet, a 2-party adaptor signature scheme can be used, e.g., in the generalized channel construction, in order to reduce the communication complexity with the blockchain. More precisely in Fig. 2.3, to spend the output of $\mathtt{TX_f}$, instead of requiring two signatures under Alice's and Bob's public keys, it is possible to require only a single signature under the combined public key of Alice and Bob. Therefore, we asked ourselves the following question:

Is it possible to extend the existing definitions to model two-party adaptor signatures and instantiate them generically?

We provide some preliminaries in Sec. 3.1 and then present our results in Sec. 3.2.

3.1. Preliminaries

First, let us recall the definition of identification schemes according to [106]. In practice, an identification schemes is a three round protocol executed between a prover and a verifier. It consists of 4 algorithms IGen, P_1, P_2 and V. The algorithms IGen, P_1 and P_2 are executed by the prover and V is executed by the verifier. Before executing the protocol, the prover generates a secret and public key pair (sk, pk) using the key generation algorithm IGen and publishes its public key pk. The goal of the prover is to convince the verifier that he or she indeed has a secret key sk corresponding to pk. The protocol is now executed as follows (see also Fig. 3.1):

1. The prover first has to generate a commitment and state (R, St) using its secret key by executing the algorithm $\mathsf{P}_1(sk)$ and sends the commitment R to the verifier (R is also called the public randomness).

2. The verifier generates a random challenge h and sends it to the prover.
3. The prover now generates a response s by executing $\mathsf{P}_2(sk, R, h, St)$ and sends it to the verifier.
4. The verifier checks if this response is valid by executing $\mathsf{V}(pk, R, h, s)$. If V returns 1, the verification succeeds, otherwise it returns 0

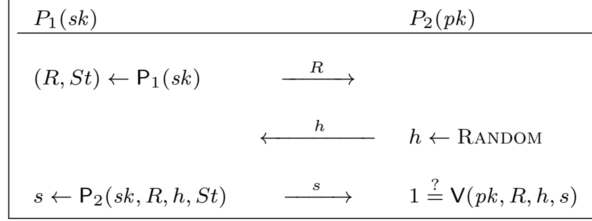

Figure 3.1.: Identification scheme protocol.

Using the Fiat-Shamir heuristic [77], it is possible to transform an Identification scheme as described above, into a signature scheme. More precisely, instead of relying on the verifier to send the challenge h, it is generated by hashing the public randomness R and message m that is going to be signed as presented in Fig. 3.2.

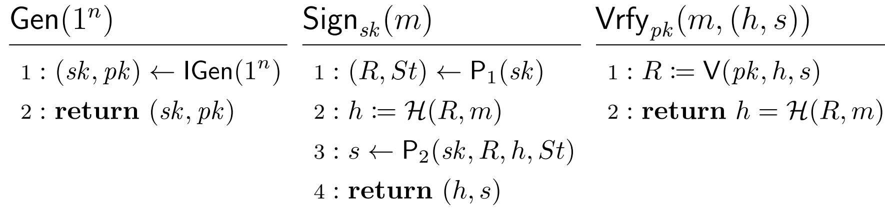

Figure 3.2.: Digital signature schemes from identification schemes [106].

3.2. Our Contribution

Motivated by the generalized channel construction, our main goal is to define and design two-party Adaptor Signature schemes. At the same time we investigated the properties that Schnorr signature satisfies which allowed transforming it into an adaptor signature. We take the following three steps as shown in Fig. 3.3, (a) we

show how to build adaptor signatures from ID schemes, then (b) we continue by showing how to build two-party signatures from ID schemes, and finally (c) we show how to build two-party adaptor signature schemes from $\mathsf{SIG}_2^{\mathsf{ID}}$. Let us explain each step in a bit more details:

Adaptor signatures from ID schemes. Our first goal is to determine if we can build adaptor signatures from a broad class of signatures. We start by showing that unique signatures, such as BLS [34], cannot be transformed into adaptor signatures. We know how to build an adaptor signature from Schnorr signature which is a signature scheme built from an Identification Scheme (ID) [106, 158]. Therefore, we focus on transforming signature schemes from ID schemes (denoted by $\mathsf{SIG}^{\mathsf{ID}}$) to adaptor signatures. We show that $\mathsf{SIG}^{\mathsf{ID}}$ schemes that satisfy certain homomorphic properties (which we will define later) can generically be transformed into adaptor signatures. We denote such an adaptor signature scheme by $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$, where R is the hard relation used in this scheme.

Two-party signatures from ID schemes. As a middle step, we show how to generically build two-party signatures with aggregatable public keys (denoted by $\mathsf{SIG}_2^{\mathsf{ID}}$) from ID schemes. Such a scheme allows two parties to jointly generate a signature which is valid under their aggregated public key. The parties can however generate their secret keys separately (non-interactively). This is in contrast to other schemes which require an interactive key generation mechanism [81]. We provide a generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ schemes that satisfy certain aggregation properties (which we will define later) to $\mathsf{SIG}_2^{\mathsf{ID}}$ schemes.

Two-party adaptor signature schemes from $\mathsf{SIG}_2^{\mathsf{ID}}$. We show how to generically transform our $\mathsf{SIG}_2^{\mathsf{ID}}$ schemes, to two-party adaptor signature schemes with aggregatable public keys, denoted by $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$. To instantiate this novel cryptographic primitive, we use similar techniques to the ones we used previously for building $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ and $\mathsf{SIG}_2^{\mathsf{ID}}$. A $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ does not require parties to generate their secret keys interactively just like a $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme.

Our concrete instantiations. Finally, we show that Schnorr, Katz-Wang and Guillou-Quisquater signature schemes [87, 101, 158] satisfy all the properties that we require in our transformations and hence can be transformed into $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$, $\mathsf{SIG}_2^{\mathsf{ID}}$ and $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ schemes. We note that Schnorr is one of the most well known and widely used signature schemes. It is also recently added to Bitcoin [156]. Katz-Wang is quite similar to Schnorr however it achieves better security. Finally, Guillou-Quisquater is quite different from the other two signature schemes as it relies on the RSA assumption, while the other schemes rely on of the Diffie-Hellman assumption (and consequently the hardness of discrete logarithm assumption). We

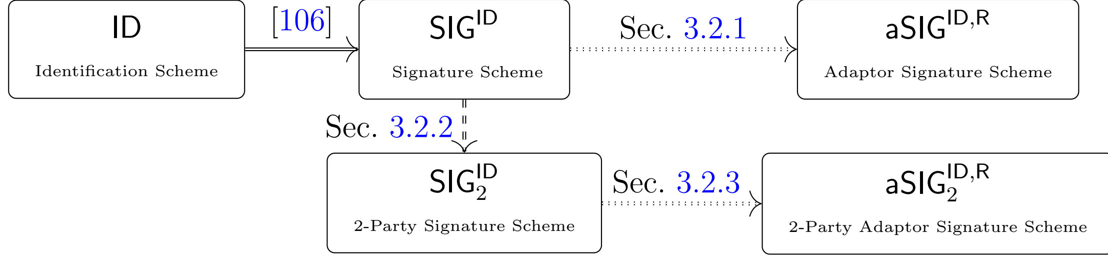

Figure 3.3.: Overview of our results. A full arrow represents a generic transformation, a dotted/dashed arrow represent a generic transformation that requires additional homomorphic/aggregation properties.

will not describe here what these assumptions are, and refer the reader to [87, 101, 158] for more information. Hence, our results show that we can construct adaptor signatures from different signature schemes based on different assumptions.

Our results have been disseminated in the following publication.

[70] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi. "Two-Party Adaptor Signatures from Identification Schemes". In: *PKC 2021, Part I*. Ed. by J. Garay. Vol. 12710. LNCS. Springer, Heidelberg, May 2021, pp. 451–480. DOI: 10.1007/978-3-030-75245-3_17. Appendix C.

3.2.1. From $\mathsf{SIG}^{\mathsf{ID}}$ Schemes to $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$

In this section, we will focus on designing an adaptor signature scheme from signature schemes based on identification schemes. But first, let us argue why unique signatures such as [34, 84, 117, 160] cannot be transformed into adaptor signatures. See Sec. 2.3.1 for the definition of (single party) adaptor signatures.

Impossibility of constructing unique adaptor signatures. In a unique signature scheme, signing the same message multiple times results in the same signature. We prove that it is impossible to construct a unique adaptor signature by contradiction, i.e., we assume such a scheme exists. This means that adapting a pre-signature on message m to a full signature using the witness y, and signing the message m results in the same signature. More precisely, for $(y, Y) \in R$ we have $\mathsf{Adapt}(\mathsf{pSign}_{sk}(m, Y), y) = \mathsf{Sign}_{sk}(m)$. We also know that one can extract the witness given a pre-signature and the corresponding full signature. Therefore, given a pre-signature $\tilde{\sigma} = \mathsf{pSign}_{sk}(m, Y)$ we can conclude:

$$y = \mathsf{Ext}(\mathsf{Adapt}(\tilde{\sigma}, y), \tilde{\sigma}, Y) = \mathsf{Ext}(\mathsf{Sign}_{sk}(m), \tilde{\sigma}, Y)$$

Hence, a signer can break the hardness assumption of the relation R. More precisely, the signer generates a pre-signature for message m on a given statement Y and also generates a signature on message m independently and executes the Ext on these values to generate y. This is a contradiction as computing y given only Y should have been hard.

We make the following observation from the above impossibility result:

The statement and witness of the hard relation must be incorporated in the randomness used for generating the (pre-)signature.

With this observation, let us see how signature schemes from identification schemes can be transformed into adaptor signatures. Our main intuition is that to generate the pre-signature, one must *shift* the public randomness R during the Sign procedure using the statement Y. We can then "adjust" the resulting pre-signature using the corresponding y (i.e., $(Y, y) \in \mathsf{R}$), in order to obtain a regular (or full) signature. More precisely we observe that $\mathsf{SIG}^{\mathsf{ID}}$ schemes as presented in Fig. 3.2, use two type of randomness, a public randomness R and a private one St. Since R can be re-calculated by the verifier using the procedure V, it is called the public randomness. St on the other hand, is only known to the signer. While R is shifted using Y to generate the pre-signature, adapting the pre-signature using y would shift St accordingly such that the resulting final signature is valid. Finally, it should be possible to extract a witness given the pre-signature and the full-signature. More precisely, to construct the pSign and pVrfy procedures, the Sign and Vrfy from Fig. 3.2 are modified as follows (formally presented in Fig. 3.4):

$\mathsf{pSign}_{sk}(m, Y)$: Shift the randomness R using the statement Y (e.g., by multiplying R and Y), and then apply the Fiat-Shamir heuristic. More precisely, the input of the hash function $\mathcal{H}$ is the shifted randomness and the message m. This procedure outputs a pre-signature $(h, \tilde{s})$.

$\mathsf{pVrfy}_{pk}(m, Y, (h, \tilde{s}))$: Similar to the pSign procedure, before checking if $h = \mathcal{H}(R, m)$, R is shifted using the statement Y.

For our transformation presented in Fig. 3.4, we define three functions f_{shift}, f_{adapt} and f_{ext}. f_{shift} shifts R using Y, f_{adapt} transforms a pre-signature value $\tilde{s}$ using y to a full signature value s, and finally f_{ext} extracts y given $\tilde{s}$ and s.

Our transformation from Fig. 3.4 cannot transform arbitrary $\mathsf{SIG}^{\mathsf{ID}}$ schemes into a secure adaptor signature w.r.t. an arbitrary hard relation. However, we show that if one can define the functions f_{shift}, f_{adapt} and f_{ext} for a $\mathsf{SIG}^{\mathsf{ID}}$ scheme and hard relation R such that the following two properties are satisfied, it is possible to transform this signature into an adaptor signature scheme.

$\mathsf{pSign}_{sk}(m, Y)$

1 : $(R_{\mathsf{pre}}, St) \leftarrow \mathsf{P}_1(sk)$
2 : $R_{\mathsf{sign}} := f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y)$
3 : $h := \mathcal{H}(R_{\mathsf{sign}}, m)$
4 : $\tilde{s} \leftarrow \mathsf{P}_2(sk, R_{\mathsf{pre}}, h, St)$
5 : **return** $(h, \tilde{s})$

$\mathsf{pVrfy}_{pk}(m, Y, (h, \tilde{s}))$

1 : $\widehat{R}_{\mathsf{pre}} := \mathsf{V}(pk, h, \tilde{s})$
2 : $\widehat{R}_{\mathsf{sign}} := f_{\mathsf{shift}}(\widehat{R}_{\mathsf{pre}}, Y)$
3 : $b := (h = \mathcal{H}(\widehat{R}_{\mathsf{sign}}, m))$
4 : **return** b

$\mathsf{Adapt}_{pk}((h, \tilde{s}), y)$

1 : $s = f_{\mathsf{adapt}}(\tilde{s}, y)$
2 : **return** (h, s)

$\mathsf{Ext}_{pk}((h, s), (h, \tilde{s}), Y)$

1 : **return** $f_{\mathsf{ext}}(s, \tilde{s})$

Figure 3.4.: Generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to an $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ scheme.

Adapt then extract is equivalent to extract then shift. This is a homomorphic property that relates the functions f_{shift}, f_{adapt}, V and the hard relation R. It states that for all statement witness pairs $(Y, y) \in \mathsf{R}$ the following two cases are equivalent: (1) first extracting the public randomness from $\tilde{s}$ via V and then applying f_{shift} to shift the public randomness by Y, and (2) first executing f_{adapt} to shift $\tilde{s}$ using y and then extracting the public randomness using V. More precisely, it must hold that:

$$f_{\mathsf{shift}}(\mathsf{V}(pk, h, \tilde{s}), Y) = \mathsf{V}(pk, h, f_{\mathsf{adapt}}(\tilde{s}, y))$$

where $(Y, y) \in \mathsf{R}$ and $(h, \tilde{s}) = \mathsf{pSign}_{sk}(m, Y)$ according to the description in Fig. 3.4.

This property is needed to prove that after correctly adapting a pre-signature into a full signature, executing the verification procedure on this adapted pre-signature returns 1.

f_{ext} and f_{adapt} are the inverse of one another. This property states that if we adapt a pre-signature to a full signature using f_{adapt} and the corresponding witness y, we would extract the same witness y after applying f_{ext} on the signature and pre-signature. More precisely, we have:

$$y = f_{\mathsf{ext}}(f_{\mathsf{adapt}}(\tilde{s}, y), \tilde{s})$$

where $(Y, y) \in \mathsf{R}$ and $(h, \tilde{s}) = \mathsf{pSign}_{sk}(m, Y)$ according to the description in Fig. 3.4.

We do not provide the proof that our transformation results in a secure adaptor signature here and refer the reader to [70] (or Appendix C) for more details. More precisely we prove the (formal version of the) following Theorem:

Theorem 3.2.1 (Informal)**.** *Assume that $\mathsf{SIG}^{\mathsf{ID}}$ is a secure signature scheme, let $f_{\mathsf{shift}}, f_{\mathsf{adapt}}$ and f_{ext} be functions satisfying the homomorphic properties explained above, and R be a hard relation. Then the $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ scheme constructed using our transformation is a secure adaptor signature scheme.*

3.2.2. From $\mathsf{SIG}^{\mathsf{ID}}$ Schemes to $\mathsf{SIG}_2^{\mathsf{ID}}$

On a high level, a two-party signature scheme with aggregatable public keys, or for short SIG_2, allows two parties to generate a single signature which can be verified under their aggregated public keys. Consider an application where two parties have to separately sign a message, e.g., a transaction which is being posted on the blockchain. By using a SIG_2, instead of submitting two signatures, the parties can jointly generate a single signature which is valid under their combined public key. This reduces the overall communication needed to authenticate a message, which is important for blockchain applications. The main difference between this primitive and a two-party threshold signature scheme is that parties generate their secret and public keys independently (non-interactively). Assume that we have two parties P_1 and P_2. On a high level, in our transformation the signing process has the following three steps (see also Fig. 3.5):

Ecxhange randomness: Both parties execute a secure random exchange protocol (see [70] for a full description). At the end of this procedure, party P_i receives two public values R_1 and R_2 (where R_i is generated by the party P_i), and the private value St_i were $i \in \{1, 2\}$. Both parties compute the combined randomness using a function $f_{\mathsf{com\text{-}rand}}$ as $R = f_{\mathsf{com\text{-}rand}}(R_1, R_2)$

Generate partial signatures: Each party signs the message independently using its secret key and the combined randomness that was agreed on during the previous step and outputs a "partial" signature (h, s_i).

Combine partial signatures: The two partial signatures are combined into a full signature using the function $s = f_{\mathsf{com\text{-}sig}}(s_1, s_2)$. The final signature is (h, s).

As in the previous section, our transformation is not generic. However, on a high level, if the $\mathsf{SIG}^{\mathsf{ID}}$ scheme satisfies the following properties, it can be transformed into a SIG_2 using the above transformation:

Combined signatures are valid under the combined public key. There exists a function $f_{\mathsf{com\text{-}pk}}$ that combines the public keys as $pk = f_{\mathsf{com\text{-}pk}}(pk_1, pk_2)$. Furthermore, the combined signature (h, s) is valid under this pk, i.e., $1 = \mathsf{Vrfy}_{pk}(m, (h, s))$.

Combined signatures are decomposable. Using the secret key of a party sk_i the full signature can be "decomposed", i.e., there exists a function $f_{\mathsf{dec\text{-}sig}}$ such that $(h, s_{3-i}) = f_{\mathsf{dec\text{-}sig}}(sk_i, pk_i, (h, s))$ where $i \in \{1, 2\}$ and $s = f_{\mathsf{com\text{-}sig}}(s_1, s_2)$.

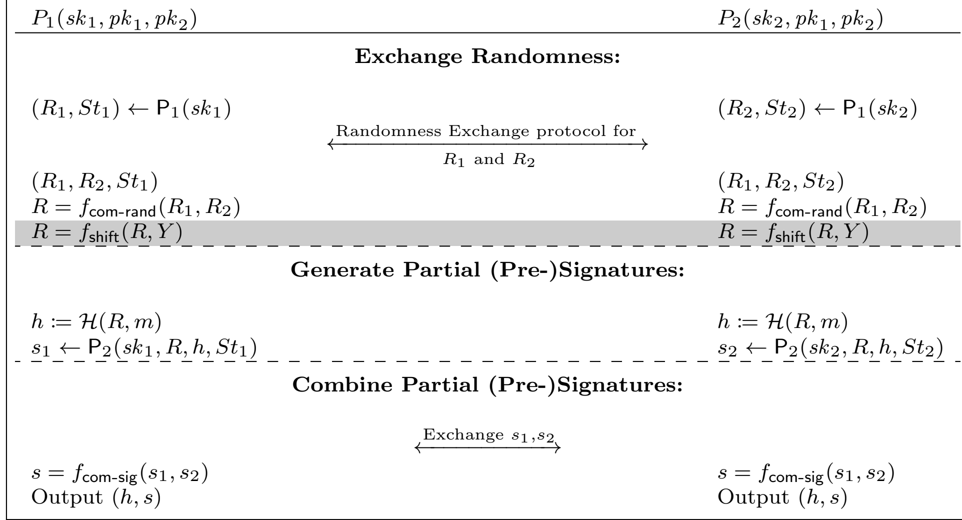

Figure 3.5.: Two-party (pre-)signing protocol. In the pre-signing protocol, the combined randomness must be shifted as shown in the gray row.

The first property is very intuitive. Naturally, we expect that the combined signature is valid under the combined public key. The second property is only needed in our security proof. We refer the reader to [70] (or Appendix C) for more information regarding our model and security proof of $\mathsf{SIG}_2^{\mathsf{ID}}$ schemes. More precisely we prove the (formal version of the) following Theorem:

Theorem 3.2.2 (Informal)**.** *Assume that $\mathsf{SIG}^{\mathsf{ID}}$ is a secure signature scheme. Further, assume that the functions $f_{\text{com-sig}}$, $f_{\text{com-pk}}$ and $f_{\text{dec-sig}}$ satisfy the aggregation properties explained above. Then the $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme constructed using our transformation is a secure two-party signature scheme.*

3.2.3. From $\mathsf{SIG}_2^{\mathsf{ID}}$ Schemes to aSIG_2

We are finally ready to present the main ideas behind our two-party adaptor signature scheme. In a nutshell, we only need to combine both transformations, i.e., the transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ and from $\mathsf{SIG}^{\mathsf{ID}}$ to $\mathsf{SIG}_2^{\mathsf{ID}}$. As shown in Fig. 3.5, the only change needed to construct aSIG_2 is to shift the randomness R before generating the partial pre-signatures. The procedures Adapt and Ext are non-interactive and remain as described in Fig. 3.4. Naturally, this transformation only works if all the required properties for the transformations in Sec. 3.2.1 and Sec. 3.2.2, hold, i.e., (1) adapt then extract is equivalent to extract then shift, (2) f_{ext} and f_{adapt} are the inverse of one another, (3) combined signatures are valid under the combined public key, and (4) combined signatures are decomposable.

The security property that a two-party adaptor signature must achieve is very similar to the properties mentioned in section Sec. 2.3.1. However, the main challenge of proving that our transformation is secure comes from the fact that one of the parties can behave maliciously. In our model, we consider an adversary that can corrupt and control one of the parties. This gives the adversary additional information (i.e., one of the secret keys) and more power as it can now communicate with the honest party. We show that if such an adversary can break the security of our scheme, one can build another adversary (sometimes called the simulator) that internally uses this adversary to break the security of the $\mathsf{SIG}^{\mathsf{ID}}$ scheme. The main challenge of constructing such a simulator is the fact that it must simulate the responses of the honest party for the adversary while using it to break the security of the $\mathsf{SIG}^{\mathsf{ID}}$ scheme. Nevertheless, we show in our paper [70] that the schemes built using our transformation are secure even in the presence of such a strong adversary. More precisely, we prove the (formal version of the) following theorem:

Theorem 3.2.3 (Informal)**.** *Assume that* $\mathsf{SIG}^{\mathsf{ID}}$ *is a secure signature scheme. Let* $f_{\mathsf{shift}}, f_{\mathsf{adapt}}$ *and* f_{ext} *and* $f_{\mathsf{com\text{-}sig}}, f_{\mathsf{com\text{-}pk}}$ *and* $f_{\mathsf{dec\text{-}sig}}$ *be functions that satisfy all the 4 properties mentioned above. Further, assume* R *is a hard relation. Then the* $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ *scheme constructed using our transformation is a secure two-party adaptor signature scheme.*

We refer the reader to [70] (or Appendix C) for more details on our models, security proofs. We further show that Schnorr, Katz-Wang and Guillou-Quisquater signatures can be transformed into $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$, $\mathsf{SIG}_2^{\mathsf{ID}}$ and aSIG_2 schemes, via our transformations.

3.3. Related Work

We have already mentioned many of the adaptor signature related work in Chapter 2. Here, we will mention a few more works related to our paper [70]. After the publication of our paper at PKC 2021, the paper [153] was posted on the eprint archive. The authors aimed to build (single-party) adaptor signature schemes from ID schemes. They further considered blind and linkable ring adaptor signatures. In a blind signature scheme [80], the signer does not know in advance the message that he or she is signing. In a linkable ring signature [167, 179], the identity of the signer remains anonymous in a group of parties, but if the signer generates two signatures, these signatures can be linked together. This type of signature is used in Monero [130] which is a privacy preserving cryptocurrency. However, the authors of [153] withdrew their paper from eprint on 26.07.2021. Very recently,

Qin et al., [154] defined blind adaptor signatures and provided an instantiation based on ECDSA signatures to construct privacy preserving PCHs. Also very recently, Dai et al., [46] extend our (single-party) definitions to achieve stronger security by returning multiple pre-signatures on the message for which the adversary wishes to return a forgery. Furthermore, they presented a generic transformation from any signature and hard relation to an adaptor signature scheme. However, their transformation requires changing the underlying signature scheme which is counterintuitive, as using the adaptor signatures resulting from their transformation would require making changes to the underlying blockchain.

Multi-Signatures. Multi-Signatures have been analyzed in numerous papers. They allow a set of parties to jointly generate a signature which can be verified under the public key of all signers. Some of the first papers describing this concept and constructions of Multi-Signatures are [88, 134, 135]. After the introduction of Multi-Signatures, the concept of Multi-Signature schemes with aggregatable public keys gained more popularity. In these schemes, the verifier only needs to use a single key to verify the final signature. When considering Multi-Signatures there are two methods for modeling their security, (1) the Knowledge of the Secret Key (KOSK) model [32], and (2) the plain public-key model [21] (or key-verification model [57]). Our transformations are also proven secure in the KOSK model. Without going into detail, in the KOSK model, when proving the security of a scheme we can assume knowledge of the secret key(s) of the corrupted parties. The plain model does not assume this additional information when proving the security of the scheme. We refer the reader to our full paper [70] for a more detailed comparison between these two models, nevertheless, the plain public-key model considers a stronger setting, as schemes secure in the plain model are also secure in the KOSK model. On the other hand, security proofs in the plain model are much more complex and prone to errors. As an example, Drijvers et al., [57] showed that the works [18, 118, 122, 168] that were proven in the plain public-key model are in fact insecure in this model. One of these works called MuSig by Maxwell et al., [122] was later fixed by Nick et al., [133].

Identification Schemes and ID-Signatures. As explained before, an ID scheme is a three round protocol that allows a party to authenticate itself to another party. There are many examples of ID-schemes such as [25, 37, 77, 82, 83, 87, 101, 125, 136, 138, 149, 158, 166]. These schemes can be transformed to non-interactive signature schemes via the Fiat-Shamir transformation [77]. Abdalla et al., [4] formalized this type of identification schemes and named them *canonical identification schemes*. In a more recent work, Kiltz et al., [106] analyzed the security of signature schemes that are built by applying the Fiat-Shamir transformation in the multi-user setting,

i.e., where the attacker who is trying to forge a signature can interact with many users and receive signatures under different public keys.

3.4. Discussion and Future Work

In this work, we formalized two-party adaptor signatures with aggregatable public keys for the first time, presented a generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ schemes to $\mathsf{aSIG}^{\mathsf{ID,R}}$, $\mathsf{SIG}_2^{\mathsf{ID}}$ and $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ schemes, and showed that Schnorr, Katz-Wang and Guillou-Quisquater schemes are compatible with our transformations. Our results can be seen as an important step towards understanding adaptor signatures and extending the original definition for more use cases, e.g., improving the efficiency of generalized channels scheme.

One immediate future work direction is to prove the security of our schemes in the plain public-key model instead of the KOSK model. Perhaps using techniques from [133] would allow not only to prove security in this model, but also reduce the communication complexity of the pre-signature procedure and consider multi-party instead of two-party adaptor signatures.

Another direction for future work is finding applications of adaptor signatures outside of the blockchain area. In adaptor signatures, the final signature, after adapting a pre-signature, looks and verifies identically to a standard signature. This feature makes them quite versatile for blockchain related applications as it allows parties to exchange a secret value when a signature is posted on-chain. However, we are not aware of any other setting outside the blockchain area that would benefit from utilizing adaptor signatures.

Finally, one can consider threshold adaptor signatures. By threshholdizing this primitive, parties will generate n key shares, while using t of them is enough to generate a signature/pre-signature. This allows for better security as an adversary who has access to $t-1$ key shares cannot forge a signature/pre-signature. This construction can then be used to design a threshold adaptor wallet scheme [7, 51, 52, 74] to store the secret keys securely and achieve better usability.

4. Lower Bounds for Plasma Protocols

Payment channels, payment channel networks, and payment channel hubs helped improve the scalability of blockchains such as Bitcoin and Ethereum. Yet we saw in the previous sections that if two parties are not directly connected to each other via a channel, they have to route their transactions through intermediaries. To guarantee that after receiving coins from the sender, the intermediary does not simply disappear and abort the transfer to the receiver, the intermediary needs to lock collateral equal to the amount being transferred. This makes these solutions rather unattractive for hubs who wish to facilitate the payment between multiple parties (see the example in Sec. 1.1.2).

To tackle the issues mentioned above, Poon and Buterin put forth an innovative idea in [151] called Plasma, to reduce the amount of collateral needed for the intermediary. In the context of Plasma protocols, the intermediary is called the *operator*. The role of the operator is only facilitating the transfer of coins between parties and (ideally) it should not lock any collateral to do so. The main idea behind Plasma protocols is to sacrifice payment channels' instant finality property, in order to reduce the amount of collateral that the operator needs to lock.

Note that Plasma protocols are only deployed on Turing complete blockchains such as Ethereum. As we will see shortly, this is mainly because the logic needed to exit users' funds from the system and use them on-chain is rather complex. In other words, one must use a smart contract to verify if a party is indeed behaving honestly and how many coins he or she owned off-chain.

The Landscape of Plasma. Shortly after the initial groundbreaking idea of Plasma payment systems [151], many new variants of Plasma protocols were proposed online [38, 40, 102, 129, 131, 142, 144, 174]. Most of these works were proposed, and sometimes even implemented, by a community of Ethereum and blockchain enthusiasts. This community typically shares their results via "white-papers", blog articles, or posts on discussion forums (such as the "Ethereum Research Forum" [76]). Although this is quite useful to discuss new ideas, using these communication channels resulted in a plethora of different proposals where

one cannot easily identify the differences between them. In Fig. 4.1, one member of the community tried to gather and mention the different Plasma proposals at the time. Seeing all these different variants of plasma protocols one might ask:

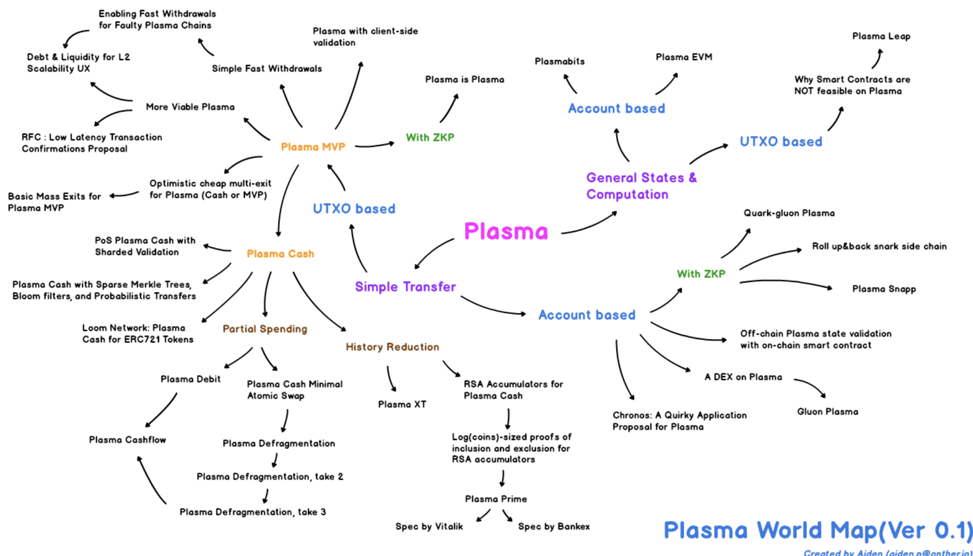

Figure 4.1.: The "Plasma World Map", illustrating the different flavors of Plasma protocols posted on Ethereum Research Forum [145].

Is it possible to design a "perfect" Plasma protocol that achieves the best properties of all the proposals?

Before answering this question let us provide some preliminaries.

4.1. Preliminaries

First, we briefly recall how the Ethereum blockchain can be modeled and present a primitive called Merkle trees which are used to build Plasma protocols.

Modeling Ethereum Blockchain. In works such as [59, 63], the authors model the blockchain as a trusted party who keeps track of the users' balances and can execute programs called smart contract. Parties can send transactions to the blockchain and transfer their coins to other parties or the smart contracts registered on the blockchain. In order for the blockchain to determine the authenticity of a transaction, users need to sign their transactions. In practice, each party is

identified via their public signing key. However, for simplicity we mention the parties' names instead. Furthermore, no user can transfer more coins than they already own. Any user can register a new smart contract on the blockchain and the logic implemented in this program determines how the coins it receives would be later redistributed. Users can also call a function of a smart contract by sending a transaction to the blockchain and specifying the smart contract, the function they wish to call and its input. In other words, each smart contract and function of a smart contract can be uniquely identified.

Merkle Trees. A Merkle tree [124] is a data structure that allows storing a list of inputs and easily prove the membership of an element in the list. In Fig. 4.2, a Merkle tree storing a list of 8 elements $x_0, \cdots, x_7$ is depicted. By $\mathcal{H}(x_0, x_1)$ we mean that x_0 and x_1 are concatenated together, and the result is hashed. These elements are called the *leaves* of Merkle tree. The idea is to hash the elements in

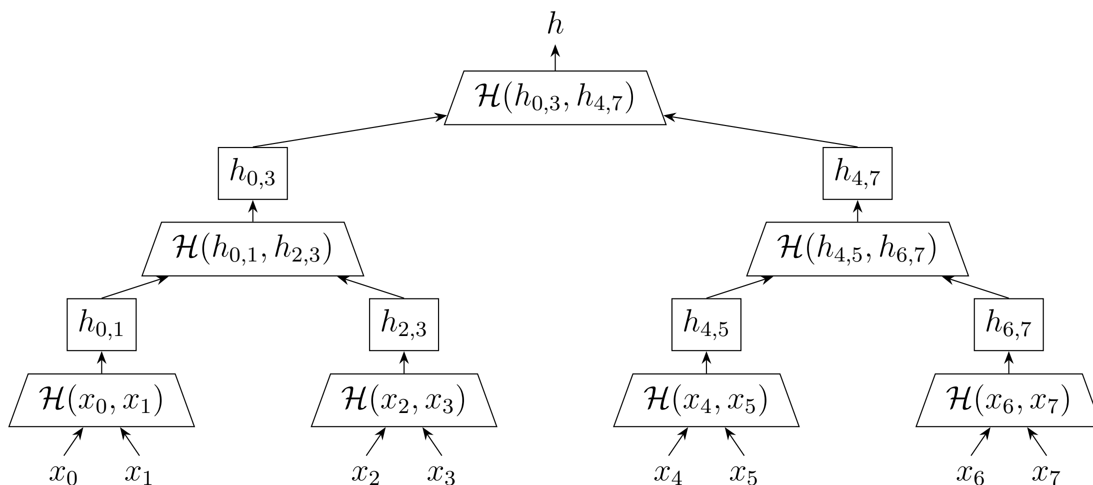

Figure 4.2.: A Merkle tree storing 8 elements $x_0, \cdots, x_7$.

this list pairwise. These hash values are then again hashed pairwise. This pairwise hashing continues until only one element is left. The final hash value is called the *root* of the Merkle tree (e.g., in Fig. 4.2 h is the root of the Merkle tree). One can prove that an element was included in a Merkle tree given the root of this tree. The size of this proof is logarithmic in the number of leaves. We will not mention how this proof can be constructed and verified here and refer the reader to [124].

4.2. Our Contribution

In this chapter, we show that one cannot construct the "perfect" Plasma protocol. Our work can also be seen as "bringing order to the huge landscape of Plasma". We categorize Plasma protocols into two main categories, provide a model for Plasma protocols and present two instantiations of Plasma protocols (one for each category) and argue that they are secure in our model. We also show that each category of Plasma protocols has their own advantages and disadvantages, and it is impossible to achieve the best of both worlds. Our work has been disseminated in the following article.

[62] S. Dziembowski, G. Fabianski, S. Faust, and S. Riahi. "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma". In: *ITCS 2021*. Ed. by J. R. Lee. Vol. 185. LIPIcs, Jan. 2021, 72:1–72:20. DOI: 10.4230/LIPIcs.ITCS.2021.72. Appendix D.

4.2.1. Our Plasma Model

We assume that the smart contract for the Plasma protocol is already deployed on the blockchain. To ease readability, we say a party sends a message to the blockchain instead of saying that it sends it to the Plasma smart contract. Furthermore, for simplicity and better readability, we avoid discussing the process of depositing coins in the Plasma system and assume that the Plasma system is initialized with n users with identities $P_1, \cdots, P_n$ and an initial balance of $x_1, \cdots, x_n$. A Plasma protocol is executed in 2 phases: (1) Transaction Phase, and (2) Exit Phase. Let us describe the high level idea behind these phases in a bit more detail:

Transaction Phase: If Alice wishes to send c coins to Bob, she signs a transaction of the form $(\text{Alice}, \text{Bob}, c)$, and sends it to the operator. At the end of this phase, the operator updates the balances of the parties locally (off-chain).

Exit Phase: In this phase, parties who wish to exit the system send their new balance to the blockchain and request an exit.

We denote the time period from the beginning of the transaction phase to the end of the exit phase as an *epoch*. Unlike payment channels, a Plasma protocol does not achieve instant finality. More precisely, in a payment channel upon the parties exchanging signatures on the new state of the channel they can post this state on-chain and receive their coins. However, in Plasma protocols parties have to wait until the end of the transaction phase for their balances to be updated.

Security in a Plasma protocol. Similar to payment and virtual channels, we need to define the notion of security for Plasma protocols. Here, we will only give a high level idea and refer the reader to [62] for the full definition. We assume that an adversary, can take control of the parties and the operator. These parties are *corrupted* and only follow the instructions of the adversary. In a nutshell, we require that an honest user with total (off-chain) balance c can exit all c coins. However, as Plasma protocols only achieve late finality a user might not be able to exit the system according to his or her latest balance, especially if the operator is malicious. Therefore, we require that an honest user is always able to exit according to her balance in either the current or previous epoch. Naturally, in order for all honest parties to be able to exit, the Plasma smart contract must have enough balance stored in it. In other words, (assuming the operator does not lock collateral) if a malicious user is able to exit the system with more balance than he or she actually has, the contract will not have enough balance to finance the exit of other honest users. Therefore, Plasma protocols require a mechanism to prevent malicious users from exiting the system and *draining* the contract's funds.

In Plasma protocols, the operator posts a short digest of the new balances to the blockchain at the end of the transaction or exit phase. This digest is sent to the blockchain in order to (1) indicate that the transaction phase is over and the balances of the parties are updated, and (2) allow the parties to later convince the blockchain how many coins they owned off-chain and exit. Most Plasma protocols use Merkle Trees [124] to store the balances of the users and the operator submits the Merkle root every epoch to the blockchain. In a nutshell, there are two ways of storing the balance of the users which results in two types of Plasma protocols, (1) Plasma MVP, where the leaves of the Merkle tree indicate how many coins does each user own in the system, and (2) Plasma Cash, where each leaf of the Merkle tree represents a coin with a fixed identity and specifies its owner. In other words, in Plasma MVP parties are mapped to their balances, while in Plasma Cash the coins are mapped to their owner. To initiate an exit, a user only needs to send the Merkle leaf and proof to the blockchain. This allows the blockchain to verify that this user owned a token in Plasma Cash or what the balance of the user was in Plasma MVP. In the next section, we will elaborate more on these two Plasma protocol categories.

4.2.2. Plasma Categories

As we briefly mentioned before, we identify two general methods of designing Plasma protocols, namely Plasma MVP [38] and Plasma Cash [40] protocols also known as Fungible and non-Fungible Plasma. The main difference between them is

how they store users' balances and how this balance can be withdrawn on-chain. In a bit more detail, in Plasma Cash parties own and exchange non-fungible tokens (tokens that have a fixed valuation and cannot be fractioned into smaller tokens) while in Plasma MVP each party has a balance and can transfer any fraction of their balance to other parties. Let us give a high-level description on how each protocol works (note that these descriptions are simplified. For the more concrete description we refer the reader to our paper [62]):

Plasma Cash. In Plasma Cash, a leaf of the Merkle tree is of the form (j, P_i) which indicates that user P_i is the current owner of leaf j (we assume that all coins have a fixed value). We now describe the Transaction and Exit Phases of Plasma Cash:

Transaction Phase: If Alice wishes to transfer a token to Bob, she signs a transaction of the form $(\mathrm{Alice}, \mathrm{Bob}, j)$ and sends it to the operator. Here j is the index of the leaf Alice wishes to transfer to Bob. At the end of this phase, the operator updates the Merkle tree, publishes the Merkle root on-chain and sends the Merkle proof of each coin to its designated user. If the Merkle tree was updated correctly Alice sends the signed tuple $(\mathrm{Alice}, \mathrm{Bob}, j)$ to Bob as a "receipt" to confirm that the transfer of token j is finalized.

Exit Phase: In this phase, parties who wish to exit the system send the Merkle proof of their coins from the previous epoch to the blockchain and indicate that they wish to exit.

Exit Challenge: If Alice tries to exit the coin she already transferred to Bob, or a coin that belongs to Bob but the operator maliciously transferred to Alice, Bob can challenge Alice by sending the signed receipt $(\mathrm{Alice}, \mathrm{Bob}, j)$ to the blockchain or asking Alice to provide the receipt that indicates Bob has transferred the ownership of this coin to Alice. In both cases, the blockchain realizes that Alice was malicious and would stop her exit.

We note that the description provided here is simplified and is mainly given as a high level overview on how Plasma Cash works.

Let us briefly explain why Plasma Cash is secure. Assume that the operator and a party, say Bob, are corrupted. If the operator maliciously transfers the ownership of Alice's coin to Bob and Bob tries to exit, Alice can challenge this Exit as mentioned above. Hence, in Plasma Cash each user only has to monitor his or her own coins and reacts only if a malicious party tries to steal them. On the other hand, if a user wishes to exit all her coins, she must exit each coin separately. This requires substantial communication with the blockchain. In the full version of

our paper [62], we prove the (the formal version) following theorem and show that a secure instantiation of Plasma Cash exists:

Theorem 4.2.1 (Plasma Cash, Informal)**.** *There exists a secure plasma protocol* Π_{Cash}*, where one honest user, when exiting, must make large on-chain communication.*

Plasma MVP. In Plasma MVP, a leaf of the Merkle tree is of the form (P_i, x_i, j) which indicates that user P_i owns x_i coins in epoch j. Let us now describe the Transaction and Exit Phases of Plasma MVP:

Transaction Phase: If Alice wishes to transfer c coins to Bob, she signs a transaction of the form $(\text{Alice}, \text{Bob}, c)$ and sends it to the operator. At the end of this phase, the operator updates the Merkle tree, publishes the Merkle root on-chain, and sends the Merkle proof of each user to them. In addition, the operator must submit the full Merkle tree to all users. If the users see any inconsistency in the final balances or do not receive the full Merkle tree they exit using their Merkle proof from the previous epoch

Exit Phase: In this phase, parties who wish to exit the system send the Merkle proof of their coins from the previous epoch to the blockchain and indicate that they wish to exit.

Let us briefly explain why the operator needs to submit the full Merkle tree to each user. Assume again that the operator and Bob, are corrupted. The operator could have potentially increased Bob's balance arbitrarily. This means that the sum of the users' balances in the leafs of the Merkle tree would be more than the total balance of the Plasma smart contract. As such, if Bob exits, there will not be enough coins left in the smart contract for all honest users to exit. That is why parties need to constantly monitor the latest balance of all users and check if the operator has cheated. There are many ways to challenge the malicious operator or parties on-chain in case of misbehavior. A naïve way would be to post the full Merkle tree on-chain and prove misbehavior to the blockchain. We will not go into detail how the challenge mechanism can be designed here and refer the reader to [38, 62, 131] for possible solutions. Nevertheless, challenge-response mechanisms for Plasma MVP are rather complex which makes designing them quite difficult. In the full version of our paper [62], we prove that a Plasma MVP exists and can be instantiated. More precisely we prove (the formal version) of the following theorem:

Theorem 4.2.2 (Plasma MVP, Informal)**.** *There exists a secure plasma protocol* Π_{MVP}*, where honest parties can be forced to make large on-chain action (mass exit) by the adversary.*

To summarize, in Plasma Cash each party owns multiple coins and can transfer the ownership of these coins to other parties. Parties only have to monitor if their coins are being maliciously withdrawn on-chain and if so can challenge this malicious exit. However, to exit the system each coin has to be withdrawn separately. In Plasma MVP, parties have a single balance and can freely transfer any fraction of their coins to other parties. To exit the system, a party only need to submit a single (short) exit message for all of his or her balances. Yet, parties need to constantly monitor the latest balance of all users and check if the operator has cheated.

4.2.3. Separation result between Plasma MVP and Cash

Let us first elaborate on the issues with Plasma MVP and monitoring the new state constantly. As long as the operator is sending the full Merkle tree to the users, they can check if their balances were updated correctly. However, as soon as the operator stops sending the new Merkle tree, parties cannot check if the operator has behaved honestly or not. This attack is known as the *Data Unavailability Attack*. In [62], we first show that data unavailability is *non-uniquely attributable*, i.e., a judge cannot decide which party is malicious, the accuser or the accused. On a high level the main reason data unavailability is non-uniquely attributable is that the operator only publishes the new state off-chain. The private off-chain communication of the parties does not have a digital footprint that can be used to prove whether the operator was malicious or not. Let us give a high level overview on how we prove this fact. Consider two scenarios:

1. The operator is malicious and does not post the latest state of the system to the users. The honest users complain to a judge (in our case the blockchain) that the operator did not post the latest state of the Plasma system off-chain.

2. The users are malicious and falsefully complain to a judge/blockchain that the operator did not post the latest state of the Plasma system.

From the point of view of the judge/blockchain, these two scenarios are indistinguishable. The judge should not accuse the users of being malicious in the first scenario. Likewise, it should not accuse the operator of being malicious in the second case. But since these two scenarios are identical from the judge's point of view we can conclude that the judge cannot attribute fault to any party. We prove this fact formally in [62].

The fact that data unavailability attack is non-uniquely attributable has a huge downside, namely that mounting this attack is "free" for the operator. In other

words, since the blockchain cannot decide if the operator was malicious or not, it also cannot punish or penalize the operator. We already saw that parties can challenge malicious exits in Plasma Cash. One might ask the question: what should the honest users of Plasma MVP do, when they face a data unavailability attack? The answer to this question is simply "take your money and run"! More precisely, parties need to leave the system immediately upon facing a data unavailability attack. We call this phenomenon, *non-uniquely attributable mass exits* as parties have to immediately exit the system before any malicious party has the opportunity to exit from the new epoch and potentially drain the smart contract (recall that if a malicious party exits with more balance than he or she actually owns at least one honest party cannot exit). Naturally, the honest parties cannot exit using their latest balance as they either have not received their Merkle proof or cannot verify if this state has been processed correctly. Plasma MVP must allow parties to exit based on their balance in the previous epoch while making sure that the parties who indeed remain in the system do not lose coins. This makes designing fungible Plasma protocols quite complex.

We show in our work [62] that it is impossible to build the perfect Plasma protocol, i.e., there does not exist a Plasma protocol with short exits that does not suffer from non-uniquely attributable mass exits. In a bit more details, we prove the (formal version of the) following Theorem:

Theorem 4.2.3 (Informal.)**.** *Let* Π *be a secure Plasma payment system with* n *users. Then either:*

1. *there exists an attack on* Π *that causes a forced on-chain action (mass exit) of large size with high (non-negligible) probability, or*
2. *there exists an attack on* Π *such that one honest user, when exiting his or her coins, must make large on-chain communication with high (non-negligible) probability.*

Moreover, both attacks have no uniquely attributable faults.

We now explain on a high level how we achieve this result. First, note that it is impossible to compress a random string arbitrarily, e.g., by hashing it, and still being able to decompress it (this is a well known fact used in different disciplines which we prove as well in [62]). In order for the blockchain to be able to determine how much balance each party had after the transaction phase of epoch i, it would need much more information than the Merkle root and the Merkle proof of a single party, as just given these values one cannot prove that the operator or the party exiting was honest. Put differently, as long as the total communication made

with the blockchain is *small*, the blockchain does not have enough information to determine what the correct balance of the parties were. This means either the total size of an exit must be large, e.g., similar to Plasma Cash, or the total communication of all honest parties with the blockchain needs to be large, e.g., similar to Plasma MVP.

To formalize the above idea, we must somehow show that the information stored and exchanged off-chain cannot be compressed into a small string. We know that a random string cannot be compressed arbitrarily without loosing some information. Hence, our idea is to build an adversary who randomly corrupts half of the parties. This represents the random string that cannot be compressed arbitrarily. Note that we assume the Plasma protocol is secure, hence the adversary should not be able to steal any of the honest users coins. However, the adversary can potentially force these parties to react on-chain in order to protect their funds.

Let us make this idea more concrete. Assume we have a Plasma system with n users each owning 1 coin. An attacker can choose, say around $n/2$ of the users at random, corrupt them, e.g., by hacking their devices, and force them to transfer their coins to Alice. At the same time the honest parties do not receive the latest state of the system, i.e., the operator is mounting a data unavailability attack. Alice should be able to exit since she indeed received all these coins honestly. The corrupted parties now act as if they neither made any transactions nor received any information from the operator, and try to exit. Naturally, if any of these corrupted parties is able to exit, an honest party will not be able to exit, and this would violate the security property of the Plasma system. Hence, the blockchain must be able to determine which user can exit. To this end, it must be able to reconstruct the list of honest (respectively corrupted) parties. However this list is indeed just a random string that the adversary choose at the beginning. Therefore, the total communication with the blockchain must be *large* in order for the blockchain to be able to reconstruct this random string. We can conclude that either Alice's exit must have a large size or all honest parties must communicate a lot of data with the blockchain (i.e., by exiting the system). This concludes the main idea behind the proof. For the full detailed proof we refer the reader to [62].

4.3. Related Work

The original idea of Plasma protocols was proposed by Poon and Buterin in [151]. They used the UTXO transaction system for their Plasma proposal due to its simplicity. However, their goal was mainly to put forth the *idea* of Plasma protocols and encourage the community to develop different variants of this protocol according

to their needs. Some examples of Plasma variants are Plasma MVP [38] and Plasma Cash [40, 109] which we mentioned in detail in this work. Plasma Debit [142] was one of the first attempts to improve Plasma Cash such that it can support more than just simple transfer of coins. Plasma Snapp [144] proposed using Zero-Knowledge proofs in order to guarantee that the operator cannot increase the balances of the parties arbitrarily. Nevertheless, the operator can still mount a data unavailability attack. One of the few formally presented Plasma solutions (which the authors call Commit-Chain protocol) are the NOCUST and NOCUST-ZKP protocols by Khalil et al. [102]. However, NOCUST requires a complex on-chain challenge-response mechanism and NOCUST-ZKP relies on zero-knowledge proofs. A less analyzed direction is executing smart contracts off-chain using Plasma. Naturally, data unavailability would be quite problematic here as users have to go on-chain to continue executing their smart contract. Nevertheless, examples of such protocols are [143, 146]. One can find many other variants of Plasma protocols on the Ethereum research forum [76], though we will not mention all proposals in this section and refer the reader to this forum.

Handling Data Unavailability. As the main issue surrounding Plasma protocols is the data unavailability attack, many works have tried to circumvent it. One direction is to use a trusted party or a committee of parties to guarantee data availability. In a nutshell, the committee members first verify if they have access to the latest state of the system and only if that is the case, the new digest (Merkle root) is valid. Examples of such works are StarkEx [164] and Oasis [171]. Another direction is simply making the data available on-chain for a short period of time in order to guarantee that all parties have access to it. This approach is called rollup and we discussed it in detail in Chapter 1.

4.4. Discussion and Future Work

In this work, we initiated the study of lower bounds for off-chain protocols. We first provided a model for Plasma protocols and presented Plasma Cash and MVP instantiations that are secure in our model. Furthermore, we showed that there is an inherent separation between Plasma Cash and MVP, i.e., Plasma protocols either suffer from non-uniquely attributable mass exits or have large exit sizes (see Table 4.1). This can help the Plasma community to focus on improving the efficiency of protocols in each of the above categories instead of trying to build a protocol which is indeed impossible to build. Furthermore, our result justifies considering different solutions such as rollup [177] to overcome this inherent limitation. In Table 4.2, we compare payment channel hubs and Plasma solutions.

As we mentioned before, payment channels suffer from high collateral while Plasma solutions do not have instant finality.

Plasma Type	Exit Size	mass exits
Plasma Cash	Large	No
Plasma MVP	Small	Yes

Table 4.1.: Plasma MVP vs Cash

Off-Chain Solution	Collateral	Instant finality
Payment Channel	Large	Yes
Plasma	Small/Non	No

Table 4.2.: Plasma vs PCHs

Let us now summarize some open problems and future works. In this work, we only focused on analyzing communication lower bounds for Plasma protocols. However, one can also consider similar lower bounds for other off-chain protocols such as payment, state, or virtual channels. Especially, solutions such as PCNs and PCHs, which involve multiple parties, can be analyzed to determine their efficiency. This can shed light on the limitations of these protocols and help design better and more efficient off-chain solutions.

Although we provided an instantiation for Plasma Cash and MVP, our protocols are not efficient and were only presented to show that these two classes of protocols exist and are secure in our model. Designing efficient and practical Plasma solutions are left out as future work. We will see one such solution in Chapter 5. Another similar direction is designing efficient Plasma protocols capable of executing smart contracts. Such protocols have not been analyzed as intensively as Plasma protocols that are only capable of making payments. Naturally, Plasma protocols capable of executing smart contracts require a more complex design to handle misbehavior and data unavailability efficiently.

Finally, Plasma protocols are only considered for blockchains that support smart contracts. Designing a Plasma or Plasma-like solution for more limited blockchains such as Bitcoin remains an open problem. In other words, analyzing the minimal assumptions needed to build Plasma protocols is an interesting open problem.

5. CommitTee

We saw in Chapter 4 how Merkle trees were used in order to design Plasma protocols. We also saw that there are two classes of Plasma protocols, Plasma Cash, and Plasma MVP. While Plasma Cash requires more communication with the blockchain when a party wishes to exit his or her coins, Plasma MVP suffers from non-uniquely attributable mass exits. We also discussed why it is impossible to build the perfect Plasma protocol.

One might however ask if it is possible to optimize Plasma MVP further and reduce the communication complexity needed in case the operator misbehaves. In addition, the fact that the operator has to periodically submit the root of the Merkle tree to the blockchain increases the cost of maintaining the Plasma payment system. In this chapter, we answer the following question:

Is it possible to design a simple yet efficient Plasma MVP protocol that can easily handle a malicious operator, assuming that the operator uses a TEE?

5.1. Our Contribution

In this section, we present *CommiTee*, an efficient Plasma protocol that uses a Trusted Execution Environment (TEE). More precisely, we show how to build a Plasma MVP protocol with a simple challenge-response mechanism to deal with a malicious operator while completely removing the necessity of sending the Merkle root every epoch to the blockchain. Our work has been disseminated in the following article.

[72] A. Erwig, S. Faust, S. Riahi, and T. Stöckert. *CommiTEE: An Efficient and Secure Commit-Chain Protocol using TEEs.* Cryptology ePrint Archive, Report 2020/1486. https://eprint.iacr.org/2020/1486. 2020. Appendix E.

5.1.1. Background on TEEs

A TEE is a chip that guarantees correct execution of the programs it runs and can store data securely. In other words, the operator of a TEE chip can neither see the secrets stored on a TEE nor influence its execution.

In our work [72], we model a TEE similar to the works by Das et al., and Pass et al. [49, 140]. A TEE is initialized with a signing keypair (msk, mpk), called master public and secret key. We assume in our work that the parties know the mpk of a TEE, e.g., they can receive it from the website of TEE's manufacturer. These keys can be used to determine if a message was generated by the program executed in the TEE or not. In other words, after installing a program inside the TEE, a user communicating with the TEE can (a) verify the program that was indeed installed on the TEE, e.g., the TEE can hash the program, sign it, and output it to the user, and (b) upon the user receiving the signed output of the program installed on the TEE, he or she can be certain that the provided output is indeed the result of the program. Note that a TEE is not a trusted party, it is just a piece of hardware that can be in the possession of a malicious party. In other words, a (potentially) malicious party can stop sending messages to the TEE and simply not send out its reply to the other parties.

The two most, prominent Examples of TEEs are Intel SGX [91, 123] and ARM TrustZone [112]. We emphasize that TEEs are not bulletproof. There has been a long line of work exposing their vulnerabilities [26, 36, 175] and also providing solutions on how to fix them [3, 161, 175]. We see this line of work orthogonal to ours and assume in our work that TEEs are secure.

5.1.2. Security and Efficiency Properties

Let us first summarize the necessary security and efficiency properties.

User Balance Security. As explained in Chapter 4, an honest user should always be able to exit her entire balance during the exit phase. As Plasma protocols do not achieve instant finality, this property essentially requires that an honest user can exit his or her balance according to either the previous or current epoch.

Efficiency. A Plasma protocol is efficient, if the duration of an epoch is independent of the number of users or transactions. This property is necessary to guarantee that malicious parties cannot increase or decrease the length of an epoch by making more transactions or creating pseudo accounts and joining the system.

5.1.3. CommiTee Protocol

The main idea behind our work is to consider an operator who has a TEE and uses it to update the balances of the users. Since a TEE is a special piece of hardware, we only assume that the operator has a TEE. We do not require users to have a TEE which is quite an important feature of our protocol.

By using a TEE, (1) the operator cannot maliciously update the balances of the users, which significantly limits the attack vector of the operator, and (2) if the honest users receive their new balance-proof, e.g., the Merkle proof of their balance, from the TEE, they are guaranteed that their balances were updated correctly. Hence, they do not need to verify the balances of the other parties and check if the operator was honest. These two advantages alone might convince us to design a new protocol where an operator uses a TEE. However, a carefully designed protocol can take advantage of the trusted hardware and optimize the communication complexity even further. In this section, we call the value returned by the TEE to the users, e.g., the Merkle proof, *balance-proof*.

Before presenting the main idea behind our protocol, let us emphasize here that having a TEE cannot prevent the non-uniquely attributable mass exits attack explained in Chapter 4. The main reason is that the operator can still stop sending the values generated by the TEE to the users. Let us analyze what can go wrong in case of a data unavailability attack. Consider a scenario where Alice wishes to send Bob 10 coins. She sends a transaction to the operator who forwards it to the TEE. At the end of the payment phase, the operator sends Bob's balance-proof to him but does not return Alice's balance-proof. In this situation, Alice can only exit using her balance-proof from the previous epoch, but Bob can exit using his balance from the latest epoch. However, if both Bob and Alice exit, 10 extra coins are withdrawn i.e., the contract will be drained. Therefore, we still need a mechanism to deal with data unavailability.

Main Idea Our main idea in [72] is as follows: We know that if any party can provide Alice's balance-proof (signed by the TEE) for the current epoch, it must have been generated by the TEE and is indeed valid. Using this balance-proof, regardless of the party providing it, Alice should be able to exit her coins. Therefore, in case of data unavailability, Alice can simply send a request to the smart contract on-chain and ask the operator to provide her balance-proof so she can exit. If the operator provides the balance-proof, Alice can exit normally. Otherwise, if the operator is non-responsive for some time the smart contract deems the operator to be malicious. In this case, the current epoch is considered to be invalid, and all parties have to exit using their balance-proof from the previous epoch.

As we can see, in case the operator is malicious we can easily force him to submit the balance-proof to the blockchain. Note that this solution would not work without being certain that the value provided by the operator was indeed generated correctly, hence this solution would not work for the Plasma MVP construction mentioned in Chapter 4 without using more advanced cryptographic primitives such as zero-knowledge proofs (for more details, see our construction from [62]).

We can now ask ourselves, is it possible to improve the efficiency of Plasma protocols even further? More specifically, can we come up with a better balance-proof mechanism than using Merkle trees and Merkle proofs?

Replacing Merkle proofs with simple signatures

As we described in the previous section, the Merkle root posted on the blockchain is used to prove that a user's balance was included in the corresponding Merkle Tree. Put differently, given the Merkle proof, the blockchain can be certain that the balance of the user was indeed produced by the operator. Yet, when using a TEE, if a message is signed by the TEE we know that it was generated honestly by the TEE. Now the question is: can we avoid using Merkle trees and publishing the Merkle root on the blockchain altogether? We answer this question positively. More precisely, the phases of the protocol are now as follows (for simplicity we do not mention how parties can deposit coins into the system):

Transaction Phase: If Alice wishes to transfer c coins to Bob, she signs a transaction of the form $(\text{Alice}, \text{Bob}, c)$ and sends it to the operator. At the end of this phase, the operator sends the list of all transactions gathered in this phase and the new blocks of the blockchain since the last epoch to the TEE. The TEE updates the balances of the users according to the off-chain transactions and on-chain exits and for each user P_i the TEE returns a signed tuple (P_i, x_i, j) where x_i is this user's balance in epoch j.

Exit Phase: In this phase, parties who wish to exit the system send their new balance tuple (P_i, x_i, j) to the blockchain and request an exit.

Exit Challenge: If a user P_i does not receive her balance-proof in this epoch, she sends a message of the form $(P_i, \text{Exit-Challenge})$ to the blockchain. The operator has a limited time, say Δ rounds, to respond:

Responsive Operator: If the operator posts the message (P_i, x_i, j) signed by the TEE, the blockchain returns x_i coins to P_i on-chain.

Malicious Operator: Otherwise, the operator is malicious and epoch j is no longer valid. Parties have to now exit using their balance-proof from epoch $j - 1$.

As we can see, our protocol is not only simple but also very efficient. It is easy to see that user balance security holds as honest parties are able to exit either according to their balance in the current or previous epoch. Let us emphasize that our solution assumes a fixed time period for each epoch (and as such the efficiency

property holds), i.e., the blockchain knows when an epoch starts and ends. This requirement is necessary to guarantee that parties cannot cheat by submitting an older balance-proof to the blockchain. We refer the reader to our full paper [72] (or Appendix E) for more technical details and our security argumentation.

Extensions In order to make CommiTee more versatile in practice, we have extended it such that it is able to, (1) change the operator instead of forcing all parties to exit if the main operator is malicious, and (2) detect if the TEE has been compromised. We refer the reader to our full paper [72] for more details on how these extensions are indeed implemented for CommiTee.

5.1.4. Evaluation of CommiTee

We have done an experimental evaluation of our protocol to show that it is indeed more efficient than the existing solutions. We provide a short summary of our results and refer the reader to [72] for more details. Many of the Plasma projects discussed in the Ethereum research forums only provide experimental evaluations and in many cases do not provide all the features of Plasma protocols. We chose the omiseGO and loom network [115, 137] implementation of Plasma MVP and Cash. Indeed, our results show that CommiTee is 2 to 16 times cheaper in terms of communication complexity when compared to Plasma MVP and Cash.

The most well known Commit-Chain solution to date is NOCUST and NOCUST-ZKP [102]. NOCUST relies on a challenge response mechanism in case the operator tries to cheat while NOCUST-ZKP relies on a zero-knowledge proof to make sure that the operator has processed all transactions correctly. Our results show that NOCUST's and NOCUST-ZKP's on-chain costs for finalizing an epoch (e.g., on-chain fees for sending the Merkle root or zero-knowledge proof), are almost 3 and more than 19 times higher than CommiTee respectively. Note that the CommiTEE's smart contract still needs to be activated (which requires some fees to be paid) at the end of an epoch to update the epoch number and announce the current epoch as concluded. Hence, although no message needs to be sent by the operator to the smart contract, the contract still needs to do some bookkeeping to start a new epoch.

5.2. Related Work

We have already mentioned many works related to Plasma protocols in Chapters 1 and 4. Here, we focus on works that use TEEs to build off-chain protocols.

Das et al., [49] proposed a solution, called FastKitten, to execute smart contracts off-chain, over blockchains with limited scripting language such as Bitcoin. Similar to our solution, FastKitten relies on an operator with a TEE who can correctly execute a smart contract off-chain. However, in their solution, the operator needs to lock collateral on-chain equal to the total balance of the parties using the system. Furthermore, parties cannot dynamically join and leave the system and are fixed at the beginning of the protocol execution. A more recent work using TEEs is POSE [79]. In this work, the authors consider a pool of operators with TEEs that can be selected for the smart contract execution. This solution assumes that at least one honest operator is selected for the execution of the smart contract and as such does not require the operators to lock collateral. Furthermore, the parties can dynamically join and leave the system. Nevertheless, these properties are only achieved due to the usage of multiple operators each equipped with a TEE.

Another work using TEEs is Ekiden [45]. In this work, the main goal of the authors is to achieve better privacy for smart contract execution. Due to their inherent distributed nature, blockchains do not guarantee any privacy, and cannot store data safely or compute on sensitive data (e.g., medical information, auctions or bids, financial transactions). Ekiden pushes the computation necessary for smart contract execution off-chain and to a network of nodes all equipped with TEEs. By doing so, Ekiden is able to achieve better privacy. Another work similar to Ekiden is Private Data Objects (PDO) [35] which also allows parties to execute smart contracts off-chain in a privacy preserving manner. In contrast to Ekiden, PDO allows parties to decide which nodes are responsible for executing the smart contract. Another similar work was presented in [99]. The authors' goal however was to synchronize stateless TEEs, i.e., TEEs that do not have access to secure storage, via the blockchain. As such, their construction is more generic than the previous two papers. They also discuss applications of their construction such as private smart contracts execution between synchronized TEEs.

Other examples of off-chain protocols with TEEs is TEEchan and TEEchain [113, 114]. TEEchain proposes a PCN construction using TEEs in order to achieve better flexibility, efficiency and security. TEEchain allows parties to dynamically deposit coins in their channels whereas in traditional payment channels, increasing the balance of a channel requires closing and reopening a new channel. To avoid losing funds in case a TEE fails or brought offline, TEEchain uses a committee of TEEs to guarantee security. Nevertheless, this solution only aims at improving the channel solutions and works over Bitcoin. Before TEEchain, TEEchan [113] also showed how to build payment channels using TEEs. However, TEEchan lacks many of the features that TEEchain achieves like multi-hop payments, dynamic deposits. Furthermore, TEEchan channels only have a limited lifetime.

In [24], the authors proposed Tesseract, a scalable TEE based cross-chain cryptocurrency exchange system using TEEs. Town Crier [181] presents a data feed (nowadays mostly knows as oracles) system, i.e., it provides the blockchain with information that are sourced on the web. By using a TEE, Town Crier guarantees that the information relayed to the blockchain are correct. Obscuro [173] presents a Bitcoin mixer solution using TEEs. A mixer, takes multiple coins as input and outputs them in a shuffled order to designated users. Hence, it would become much harder to track coins or identify the source of a transaction. Obscuro uses TEEs to design a simple but secure mixer protocol. Hawk [110] presents an off-chain solution using TEEs to execute smart contracts in a privacy preserving manner. However, parties need to post their inputs (in an encrypted form) on-chain.

5.3. Discussion and Future Work

In this chapter, we presented a simple yet efficient fungible Plasma/Commit-Chain solution using TEEs. Unlike other fungible Plasma protocols [38], honest parties do not need to check the full state (Merkle Tree) of the Plasma system and only need to submit a single message signed by the TEE when exiting the system. Our protocol also does not rely on advanced cryptographic primitives such as zero-knowledge proofs [144]. An interesting future work direction would be to design an efficient Plasma protocol capable of executing smart contracts off-chain. Although works such as FastKitten and POSE [49, 79] are also able to execute smart contracts off-chain they either require using multiple operators or have some limitations, e.g., requiring the operator to lock collateral and having a fixed set of parties. Another direction would be to consider multiple operators who are simultaneously active and process transactions and as long as a majority of them are online (i.e., the operator does not mount a data unavailability attack) the parties receive their balance-proof. Naturally, one can execute a consensus protocol between these TEEs. However, it would be interesting to come up with more efficient and tailor made solutions. We showed how to build an efficient fungible Plasma protocol using a TEE. It would also be interesting to see if one can improve the efficiency of non-fungible Plasma, using a TEE. Finally, one can also consider a Plasma protocol that can act as a bridge between multiple blockchains, i.e., parties can deposit coins in one blockchain yet exit their coins in a different blockchain.

Impact of our work. After publishing our paper, CommiTee has been implemented by the PolyCrypt GmbH [150] under the brand name Erdstall [66].

6. Conclusion

In this thesis, we analyzed the minimal assumptions required to construct off-chain protocols. We started in Chapter 2 with generalized and virtual channel constructions over Bitcoin [11, 12]. Generalized channels are a generalization of state channels over Ethereum. They allow executing applications off-chain that are supported by the underlying blockchain even if the underlying blockchain has a limited scripting language. One can also view state channels as a special case of generalized channels that are only designed for blockchains with a Turing complete scripting language. Works such as [59, 63] put forth state channel constructions, yet these solutions can only be deployed over blockchains capable of executing smart contracts such as Ethereum. Our generalized channels on the other hand can be executed over Bitcoin. This is a huge step towards improving blockchains' and more specifically, Bitcoin's scalability. In order to design generalized channels, we used a primitive called *adaptor signatures*. We formalized adaptor signatures as a standalone primitive for the first time in [11]. This allows other works in the future to use adaptor signatures and argue the security of their scheme more easily. We further showed that the single-party Schnorr and ECDSA instantiations of adaptor signatures are secure in our model. As mentioned in Chapter 2, our work has been the building block for many followup works such as [75, 128, 169, 170].

We then showed how to build virtual channels over blockchains with a limited scripting language such as Bitcoin. A Virtual channel allows two parties who have a channel with a common intermediary, to make direct off-chain payments without requiring the intermediary to route the payments. Similar to generalized channels, the virtual channel constructions known before our work could only be deployed over blockchains that could execute smart contracts [59, 60, 63]. All in all, we showed that building generalized and virtual channels does not require the underlying blockchain to support Turing complete smart contracts, and they can be deployed over blockchains with a much more restricted scripting language.

Although recent works, such as [14, 104], have extended the functionality of our virtual channels, there are still some future work directions open for investigation. First and foremost, extending our construction from two-party to multi-party channels can be an interesting future work direction. This would allow executing not only two-party but also multi-party applications off-chain.

Another direction for future work is finding new ways to analyze the security of complex blockchain protocols. As mentioned in Chapter 2, analyzing protocols in UC is unfortunately quite cumbersome for complex protocols such as the ones deployed over blockchains. To tackle this issue, there are two complementary approaches that can be investigated. First, one can design a specific model for blockchain applications that already includes many of the machinery needed for proving security in the blockchain setting, e.g., a unified model of blockchain itself. Another direction would be to design formal/automated verification tools similar to [31, 64, 107, 157]. This would allow easier verification of the protocol's security and even simplify the process of implementing such protocols. A natural extension of this direction is analyzing if the implementation of the software and protocols are indeed secure. This would guarantee that the solutions are not only secure from a conceptual point of view but also in practice.

Finally, optimizing and calculating the duration of the time-locks are necessary in order to implement and use off-chain protocols in practice. Similarly, making sure that the parties are financially incentivised to route transactions is crucial for a functioning off-chain payment system. All in all, analyzing the financial/economic aspects of off-chain protocols is an interesting research question.

We continued our work in Chapter 3 by formalizing and designing two-party adaptor signature schemes with aggregatable public keys [70]. This scheme allows two-parties to jointly generate a pre-signature/signature which is valid under their aggregated public key. We showed how both the single, and two-party variants of adaptor signatures, and two-party signatures can be constructed (almost) generically from identification schemes that satisfy certain properties. We further showed that Schnorr, Katz-Wang, and Guillou-Quisquater schemes [87, 101, 159] satisfy these properties and hence can be generically transformed into single and two-party adaptor signatures and two-party signatures with aggregatable public keys. The two-party adaptor signature construction allows improving the efficiency of our generalized channel constructions as explained previously in Chapter 3. Finally, we showed that unique signatures such as BLS [34] cannot be transformed into adaptor signatures. Our work can be seen as the first step towards understanding adaptor signatures. A natural future work direction is extending our model and construction to multi-party adaptor signatures or analyzing if other signature schemes such as ring signatures [167, 179] can also be transferred into adaptor signatures. Furthermore, it would be interesting to prove the security of our scheme in the plain public key model [21]. As we discussed in Chapter 3, we currently only prove the security of our scheme in the KOSK model [32] which is weaker than the plain public key model. Finally, finding new settings other than the currently explored blockchain applications, where adaptor signatures can be used,

is an interesting general research direction.

By this point, we analyzed the possibility of constructing schemes and protocols over Bitcoin that were already developed for the Ethereum blockchain, i.e., generalized and virtual channels. In Chapter 4, we presented our paper [62] where we analyzed a new proposal called Plasma. Plasma protocols were introduced as an alternative to channels in order to increase the scalability of the Ethereum blockchain. The community came up with two standards, namely Plasma Cash and MVP [38, 40, 109] each with its own advantages and disadvantages. While Plasma Cash was easier to design and instantiate, parties who wish to bring their funds back on-chain need to make large communication with the blockchain. Plasma MVP on the other hand did not require large on-chain communication when parties want to leave the system. However, in case the operator cheats, and does not send the state of the Plasma MVP system off-chain to the users, every honest party has to immediately leave the system. We called this phenomenon mass exit and proved that it is non-uniquely attributable, i.e., the blockchain cannot decide if the operator behaved maliciously or not. Hence, the operator cannot be punished for causing this mass exit. We proved that one cannot construct a protocol that has the best features of both Plasma MVP and Cash. More precisely, it is impossible to build a plasma system that has short exits and does not suffer from non-uniquely attributable mass exits.

As future work, one can analyze the limitations of other off-chain protocols such as payment channels. This would allow both researchers and practitioners to know the theoretical limitations of these off-chain protocols and avoid spending time on designing protocols that are impossible to build. Another future work direction would be designing more efficient Plasma Cash and MVP protocols and proving that they are indeed secure. Finally, one can consider scenarios where instead of a single operator, a set of parties take the role of the operator. Naturally, these parties must be synchronized, however minimizing the communication between these parties and avoiding the usage of expensive generic Multi Party Computation techniques is an interesting direction for future research.

Finally, in Chapter 5 we presented an efficient yet simple Plasma MVP protocol using TEE called CommiTEE [72]. This protocol does not have many of the inefficiencies of the other Plasma protocols. CommiTEE has a much simpler challenge/response mechanism compared to other Plasma MVP constructions without relying on complex cryptographic primitives such as zero-knowledge proofs. Furthermore, our construction does not need the operator to send messages periodically to the blockchain. This is in contrast to all previous Plasma protocols. We further showed via an experimental evaluation that our construction is much more efficient than the existing implementations of Plasma/Commit-Chain protocols.

Due to its efficiency and practicallity, CommiTee has been implemented by the PolyCrypt GmbH [150] under the brand name Erdstall [66]. In this chapter, we only considered off-chain payments, however, using TEEs one might be able to extend the functionality of CommiTEE in multiple ways. First, CommiTEE can be extended to also support the off-chain execution of smart contracts. Another example would be to support multiple operators that are active at the same time and jointly process transactions. One can also analyze if using a TEE helps design a more efficient Plasma Cash protocol. Similar to payment channels, analyzing the financial aspects of the Plasma protocols and making sure that the operators are financially compensated for facilitating the off-chain payments is an interesting an important research direction. Finally, supporting payments between multiple blockchains such that parties can join from one blockchain and exit their funds in another blockchain is an interesting open problem.

7. Bibliography

[1] 1ML. *Lightning Network Statistics*. https://1ml.com/statistics. 2022. Accessed on: 2.1.2023.

[2] *A Deep Dive Into Blockchain Scalability*. https://tinyurl.com/3bj6uz5c. Accessed on: 2.1.2023.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 1 (2009), pp. 1–40.

[4] M. Abdalla, J. H. An, M. Bellare, and C. Namprempre. "From Identification to Signatures via the Fiat-Shamir Transform: Minimizing Assumptions for Security and Forward-Security". In: *EUROCRYPT 2002*. 2002, pp. 418–433.

[5] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. "Chainspace: A Sharded Smart Contracts Platform". In: *NDSS 2018*. 2018.

[6] N. Alkeilani Alkadri, P. Das, A. Erwig, S. Faust, J. Krämer, S. Riahi, and P. Struck. "Deterministic Wallets in a Quantum World". In: *ACM CCS 2020*. Ed. by J. Ligatti, X. Ou, J. Katz, and G. Vigna. ACM Press, Nov. 2020, pp. 1017–1031. DOI: 10.1145/3372297.3423361.

[7] N. Alkeilani Alkadri, P. Das, A. Erwig, S. Faust, J. Krämer, S. Riahi, and P. Struck. "Deterministic Wallets in a Quantum World". In: *ACM CCS 2020*. 2020, pp. 1017–1031.

[8] L. Aumayr, K. Abbaszadeh, and M. Maffei. *Thora: Atomic And Privacy-Preserving Multi-Channel Updates*. Cryptology ePrint Archive, Report 2022/317. https://eprint.iacr.org/2022/317. 2022.

[9] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi. *Bitcoin-Compatible Virtual Channels*. Cryptology ePrint Archive, Report 2020/554. https://eprint.iacr.org/2020/554. 2020.

[10] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostakova, M. Maffei, P. Moreno-Sanchez, and S. Riahi. *Generalized Bitcoin-Compatible Channels*. Cryptology ePrint Archive, Report 2020/476. https://eprint.iacr.org/2020/476. 2020.

[11] L. Aumayr, O. Ersoy, A. Erwig, S. Faust, K. Hostáková, M. Maffei, P. Moreno-Sanchez, and S. Riahi. “Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures”. In: *ASIACRYPT 2021, Part II*. Ed. by M. Tibouchi and H. Wang. Vol. 13091. LNCS. Springer, Heidelberg, Dec. 2021, pp. 635–664. DOI: 10.1007/978-3-030-92075-3_22. Appendix A.

[12] L. Aumayr, M. Maffei, O. Ersoy, A. Erwig, S. Faust, S. Riahi, K. Hostáková, and P. Moreno-Sanchez. “Bitcoin-Compatible Virtual Channels”. In: *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021, pp. 901–918. DOI: 10.1109/SP40001.2021.00097. Appendix B.

[13] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei. “Blitz: Secure Multi-Hop Payments Without Two-Phase Commits”. In: *USENIX Security 2021*. 2021, pp. 4043–4060.

[14] L. Aumayr, P. Moreno-Sanchez, A. Kate, and M. Maffei. *Donner: UTXO-Based Virtual Channels Across Multiple Hops*. Cryptology ePrint Archive, Report 2021/855. https://eprint.iacr.org/2021/855. 2021.

[15] L. Aumayr, S. A. Thyagarajan, G. Malavolta, P. Monero-Sánchez, and M. Maffei. *Sleepy Channels: Bitcoin-Compatible Bi-directional Payment Channels without Watchtowers*. Cryptology ePrint Archive, Report 2021/1445. https://eprint.iacr.org/2021/1445. 2021.

[16] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. “Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability”. In: *ACM CCS 2018*. 2018, pp. 913–930.

[17] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. “Bitcoin as a Transaction Ledger: A Composable Treatment”. In: *CRYPTO 2017, Part I*. 2017, pp. 324–356.

[18] A. Bagherzandi, J. H. Cheon, and S. Jarecki. “Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma”. In: *ACM CCS 2008*. 2008, pp. 449–458.

[19] W. Banasik, S. Dziembowski, and D. Malinowski. “Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts”. In: *ESORICS 2016, Part II*. 2016, pp. 261–280.

[20] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. “SoK: Consensus in the age of blockchains”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 2019, pp. 183–198.

[21] M. Bellare and G. Neven. “Multi-signatures in the plain public-Key model and a general forking lemma”. In: *ACM CCS 2006*. 2006, pp. 390–399.

[22] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474.

[23] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge”. In: *CRYPTO 2013, Part II*. 2013, pp. 90–108.

[24] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels. “Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware”. In: *ACM CCS 2019*. 2019, pp. 1521–1538.

[25] T. Beth. “Efficient Zero-Knowledge Identification Scheme for Smart Cards”. In: *EUROCRYPT’88*. 1988, pp. 77–84.

[26] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX”. In: *USENIX Security 2018*. 2018, pp. 1213–1227.

[27] *Bitcoin Cash*. https://bitcoincash.org. Accessed on: 2.1.2023.

[28] *Bitcoin Transaction Rate Per-Second*. https://www.blockchain.com/charts/transactions-per-second. Accessed on: 2.1.2023.

[29] *Bitcoin Transaction Rate Per-Second*. https://bitcoinvisuals.com/chain-tx-second. Accessed on: 2.1.2023.

[30] *Bitcoin Wiki: Payment Channels*. https://tinyurl.com/y6msnk7u. Accessed on: 2.1.2023.

[31] B. Blanchet. “Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif”. In: *Foundations and Trends® in Privacy and Security* 1-2 (2016), pp. 1–135. ISSN: 2474-1558.

[32] A. Boldyreva. “Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme”. In: *PKC 2003*. 2003, pp. 31–46.

[33] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. In: *EUROCRYPT 2003*. 2003, pp. 416–432.

[34] D. Boneh, B. Lynn, and H. Shacham. “Short Signatures from the Weil Pairing”. In: *ASIACRYPT 2001*. 2001, pp. 514–532.

[35] M. Bowman, A. Miele, M. Steiner, and B. Vavala. *Private Data Objects: an Overview*. 2018. DOI: 10.48550/ARXIV.1807.05686. URL: https://arxiv.org/abs/1807.05686.

[36] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. Vancouver, BC, 2017.

[37] E. F. Brickell and K. S. McCurley. “An Interactive Identification Scheme Based on Discrete Logarithms and Factoring”. In: *EUROCRYPT’90*. 1991, pp. 63–71.

[38] V. Buterin. *Minimal Viable Plasma*. https://ethresear.ch/t/minimal-viable-plasma. 2018. Accessed on: 2.1.2023.

[39] V. Buterin. *On-chain scaling to potentially 500 tx/sec through mass tx validation*. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477. Accessed on: 2.1.2023.

[40] V. Buterin. *Plasma Cash: Plasma with much less per-user data checking*. https://ethresear.ch/t/plasma-cash-plasma-with-much-less-per-user-data-checking/1298. 2018. Accessed on: 2.1.2023.

[41] V. Buterin and V. Griffith. "Casper the friendly finality gadget". In: *arXiv preprint arXiv:1710.09437* (2017).

[42] R. Canetti. "Security and Composition of Multiparty Cryptographic Protocols". In: *Journal of Cryptology* 1 (2000), pp. 143–202.

[43] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. 2001, pp. 136–145.

[44] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. "Universally Composable Security with Global Setup". In: *TCC 2007*. 2007, pp. 61–85.

[45] 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: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019, pp. 185–200.

[46] W. Dai, T. Okamoto, and G. Yamamoto. *Stronger Security and Generic Constructions for Adaptor Signatures*. Cryptology ePrint Archive, Paper 2022/1687. https://eprint.iacr.org/2022/1687. 2022.

[47] P. Daian, R. Pass, and E. Shi. "Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake". In: *FC 2019*. 2019, pp. 23–41.

[48] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption". In: *CRYPTO 2012*. 2012, pp. 643–662.

[49] 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: *USENIX Security 2019*. 2019, pp. 801–818.

[50] P. Das, A. Erwig, S. Faust, J. Loss, and S. Riahi. "The Exact Security of BIP32 Wallets". In: *ACM CCS 2021*. Ed. by G. Vigna and E. Shi. ACM Press, Nov. 2021, pp. 1020–1042. DOI: 10.1145/3460120.3484807.

[51] P. Das, A. Erwig, S. Faust, J. Loss, and S. Riahi. "The Exact Security of BIP32 Wallets". In: *ACM CCS 2021*. 2021, pp. 1020–1042.

[52] P. Das, S. Faust, and J. Loss. "A Formal Treatment of Deterministic Wallets". In: *ACM CCS 2019*. 2019, pp. 651–668.

[53] B. David, P. Gazi, A. Kiayias, and A. Russell. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *EUROCRYPT 2018, Part II*. 2018, pp. 66–98.

[54] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems*. Cham, 2015, pp. 3–18.

[55] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems 2015*. 2015, pp. 3–18.

[56] Y. Dong, I. Goldberg, S. Gorbunov, and R. Boutaba. "Astrape: Anonymous Payment Channels with Boring Cryptography". In: *Applied Cryptography and Network Security*. Cham, 2022, pp. 748–768.

[57] M. Drijvers, K. Edalatnejad, B. Ford, E. Kiltz, J. Loss, G. Neven, and I. Stepanovs. "On the Security of Two-Round Multi-Signatures". In: *2019 IEEE Symposium on Security and Privacy*. 2019, pp. 1084–1101.

[58] C. Dwork and M. Naor. "Pricing via Processing or Combatting Junk Mail". In: *CRYPTO'92*. 1993, pp. 139–147.

[59] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. "Multi-party Virtual State Channels". In: *EUROCRYPT 2019, Part I*. 2019, pp. 625–656.

[60] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. "Perun: Virtual Payment Hubs over Cryptocurrencies". In: *2019 IEEE Symposium on Security and Privacy*. 2019, pp. 106–123.

[61] S. Dziembowski, G. Fabiański, S. Faust, and S. Riahi. *Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma*. Cryptology ePrint Archive, Report 2020/175. https://eprint.iacr.org/2020/175. 2020.

[62] S. Dziembowski, G. Fabianski, S. Faust, and S. Riahi. "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma". In: *ITCS 2021*. Ed. by J. R. Lee. Vol. 185. LIPIcs, Jan. 2021, 72:1–72:20. DOI: 10.4230/LIPIcs.ITCS.2021.72. Appendix D.

[63] S. Dziembowski, S. Faust, and K. Hostáková. "General State Channel Networks". In: *ACM CCS 2018*. 2018, pp. 949–966.

[64] *EasyCrypt*. https://github.com/EasyCrypt/easycrypt. Accessed on: 2.1.2023.

[65] C. Egger, P. Moreno-Sanchez, and M. Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks". In: *ACM CCS 2019*. 2019, pp. 801–815.

[66] *Erdstall*. https://erdstall.dev. Accessed on: 2.1.2023.

[67] P. W. Eric Lombrozo Johnson Lau. *Segregated Witness (Consensus layer)*. https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki. 2015. Accessed on: 2.1.2023.

[68] O. Ersoy, P. Moreno-Sanchez, and S. Roos. *Get Me out of This Payment! Bailout: An HTLC Re-routing Protocol*. Cryptology ePrint Archive, Report 2022/958. https://eprint.iacr.org/2022/958. 2022.

[69] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi. *Two-Party Adaptor Signatures From Identification Schemes*. Cryptology ePrint Archive, Report 2021/150. https://eprint.iacr.org/2021/150. 2021.

[70] A. Erwig, S. Faust, K. Hostáková, M. Maitra, and S. Riahi. "Two-Party Adaptor Signatures from Identification Schemes". In: *PKC 2021, Part I*. Ed. by J. Garay. Vol. 12710. LNCS. Springer, Heidelberg, May 2021, pp. 451–480. DOI: 10.1007/978-3-030-75245-3_17. Appendix C.

[71] A. Erwig, S. Faust, and S. Riahi. *Large-Scale Non-Interactive Threshold Cryptosystems Through Anonymity*. Cryptology ePrint Archive, Report 2021/1290. https://eprint.iacr.org/2021/1290. 2021.

[72] A. Erwig, S. Faust, S. Riahi, and T. Stöckert. *CommiTEE: An Efficient and Secure Commit-Chain Protocol using TEEs*. Cryptology ePrint Archive, Report 2020/1486. https://eprint.iacr.org/2020/1486. 2020. Appendix E.

[73] A. Erwig, J. Hesse, M. Orlt, and S. Riahi. "Fuzzy Asymmetric Password-Authenticated Key Exchange". In: *ASIACRYPT 2020, Part II*. Ed. by S. Moriai and H. Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 761–784. DOI: 10.1007/978-3-030-64834-3_26.

[74] A. Erwig and S. Riahi. "Deterministic Wallets for Adaptor Signatures". In: *Computer Security – ESORICS 2022*. Ed. by V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng. Cham: Springer Nature Switzerland, 2022, pp. 487–506. ISBN: 978-3-031-17146-8.

[75] M. F. Esgin, O. Ersoy, and Z. Erkin. "Post-Quantum Adaptor Signatures and Payment Channel Networks". In: *ESORICS 2020, Part II*. 2020, pp. 378–397.

[76] *Ethereum Research Forum*. https://ethresear.ch/c/layer-2/plasma/7. Accessed on: 2.1.2023.

[77] A. Fiat and A. Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *CRYPTO'86*. 1987, pp. 186–194.

[78] L. Fournier. *One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures*. https://tinyurl.com/y4qxopxp. 2019. Accessed on: 2.1.2023.

[79] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A.-R. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *arXiv preprint arXiv:2210.07110* (2022).

[80] G. Fuchsbauer, A. Plouviez, and Y. Seurin. “Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model”. In: *EUROCRYPT 2020, Part II*. 2020, pp. 63–95.

[81] R. Gennaro, S. Goldfeder, and A. Narayanan. “Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security”. In: *ACNS 16*. 2016, pp. 156–174.

[82] M. Girault. “An Identity-based Identification Scheme Based on Discrete Logarithms Modulo a Composite Number (Rump Session)”. In: *EUROCRYPT’90*. 1991, pp. 481–486.

[83] E.-J. Goh, S. Jarecki, J. Katz, and N. Wang. “Efficient Signature Schemes with Tight Reductions to the Diffie-Hellman Problems”. In: *Journal of Cryptology* 4 (2007), pp. 493–514.

[84] S. Goldwasser and R. Ostrovsky. “Invariant Signatures and Non-Interactive Zero-Knowledge Proofs are Equivalent (Extended Abstract)”. In: *CRYPTO’92*. 1993, pp. 228–245.

[85] M. Green and I. Miers. “Bolt: Anonymous Payment Channels for Decentralized Currencies”. In: *ACM CCS 2017*. 2017, pp. 473–489.

[86] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. *SoK: Off The Chain Transactions*. Cryptology ePrint Archive, Report 2019/360. https://eprint.iacr.org/2019/360. 2019.

[87] L. C. Guillou and J.-J. Quisquater. “A “Paradoxical” Indentity-Based Signature Scheme Resulting from Zero-Knowledge”. In: *CRYPTO’88*. 1990, pp. 216–231.

[88] T. Hardjono and Y. Zheng. “A Practical Digital Multisignature Scheme Based on Discrete Logarithms”. In: *AUSCRYPT’92*. 1993, pp. 122–132.

[89] M. Hearn. *Micro-payment channels implementation now in bitcoinj*. https://bitcointalk.org/index.php?topic=244656.0. Accessed on: 2.1.2023.

[90] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. “TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub”. In: *NDSS 2017*. 2017.

[91] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. “Using Innovative Instructions to Create Trustworthy Software Solutions”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel, 2013.

[92] *Hyperledger*. https://www.hyperledger.org. Accessed on: 2.1.2023.

[93] M. Jakobsson and A. Juels. “Proofs of work and bread pudding protocols”. In: *Secure information networks*. 1999, pp. 258–272.

[94] D. Johnson, A. Menezes, and S. Vanstone. “The elliptic curve digital signature algorithm (ECDSA)”. In: *International journal of information security* 1 (2001), pp. 36–63.

[95] M. Jourenko, K. Kurazumi, M. Larangeira, and K. Tanaka. *SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies*. Cryptology ePrint Archive, Report 2019/352. https://eprint.iacr.org/2019/352. 2019.

[96] M. Jourenko, M. Larangeira, and K. Tanaka. “Lightweight Virtual Payment Channels”. In: *CANS 20*. 2020, pp. 365–384.

[97] M. Jourenko, M. Larangeira, and K. Tanaka. “Payment Trees: Low Collateral Payments for Payment Channel Networks”. In: *Financial Cryptography and Data Security*. Berlin, Heidelberg, 2021, pp. 189–208.

[98] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. “Arbitrum: Scalable, private smart contracts”. In: *USENIX Security 2018*. 2018, pp. 1353–1370.

[99] G. Kaptchuk, M. Green, and I. Miers. “Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers”. In: *NDSS 2019*. 2019.

[100] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition*. 2nd. 2014.

[101] J. Katz and N. Wang. “Efficiency Improvements for Signature Schemes with Tight Security Reductions”. In: *ACM CCS 2003*. 2003, pp. 155–164.

[102] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Report 2018/642. https://eprint.iacr.org/2018/642. 2018.

[103] A. Kiayias and O. S. T. Litos. “A Composable Security Treatment of the Lightning Network”. In: *CSF 2020 Computer Security Foundations Symposium*. 2020, pp. 334–349.

[104] A. Kiayias and O. S. T. Litos. *Elmo: Recursive Virtual Payment Channels for Bitcoin*. Cryptology ePrint Archive, Report 2021/747. https://eprint.iacr.org/2021/747. 2021.

[105] A. Kiayias, A. Russell, B. David, and R. Oliynykov. “Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol”. In: *CRYPTO 2017, Part I*. 2017, pp. 357–388.

[106] E. Kiltz, D. Masny, and J. Pan. “Optimal Security Proofs for Signatures from Identification Schemes”. In: *CRYPTO 2016, Part II*. 2016, pp. 33–61.

[107] N. Kobeissi, G. Nicolas, and M. Tiwari. "Verifpal: Cryptographic Protocol Analysis for the Real World". In: *INDOCRYPT 2020*. 2020, pp. 151–202.

[108] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *2018 IEEE Symposium on Security and Privacy*. 2018, pp. 583–598.

[109] G. Konstantopoulos. *Plasma Cash: Towards more efficient Plasma constructions*. 2019. DOI: 10.48550/ARXIV.1911.12095. URL: https://arxiv.org/abs/1911.12095.

[110] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy*. 2016, pp. 839–858.

[111] D. R. Lee, Y. Jang, and H. Kim. "Poster: A Proof-of-Stake (PoS) Blockchain Protocol using Fair and Dynamic Sharding Management". In: *ACM CCS 2019*. 2019, pp. 2553–2555.

[112] A. Limited. *Building a Secure System using TrustZone Technology*. https://tinyurl.com/amhjkfxy. 2009. Accessed on: 2.1.2023.

[113] J. Lind, I. Eyal, P. Pietzuch, and E. G. Sirer. *Teechan: Payment Channels Using Trusted Execution Environments*. 2016. DOI: 10.48550/ARXIV.1612.07766. URL: https://arxiv.org/abs/1612.07766.

[114] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch. "Teechain: A Secure Payment Network with Asynchronous Blockchain Access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville, Ontario, Canada, 2019, 63–79.

[115] *loom network*. https://github.com/loomnetwork/plasma-cash. Accessed on: 2.1.2023.

[116] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. "A Secure Sharding Protocol For Open Blockchains". In: *ACM CCS 2016*. 2016, pp. 17–30.

[117] A. Lysyanskaya. "Unique Signatures and Verifiable Random Functions from the DH-DDH Separation". In: *CRYPTO 2002*. 2002, pp. 597–612.

[118] C. Ma, J. Weng, Y. Li, and R. Deng. "Efficient discrete logarithm based multi-signature scheme in the plain public key model". In: *Designs, Codes and Cryptography* 2 (2010), pp. 121–133.

[119] madcrypto. *Cryptocurrency Statistics Almanack 2022*. https://www.madcrypto.com/statistics/?gclid=CjwKCAjwsfuYBhAZEiwA5a6CDND-nXXnH4uM7r9d90PPOzlEHBJ676ZwGAogK-gIl4U8Q5AKzPPy3hoCDBgQAvD_BwE. 2022. Accessed on: 2.1.2023.

[120] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. “Concurrency and Privacy with Payment-Channel Networks”. In: *ACM CCS 2017*. 2017, pp. 455–471.

[121] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. “Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability”. In: *NDSS 2019*. 2019.

[122] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille. “Simple schnorr multi-signatures with applications to bitcoin”. In: *Designs, Codes and Cryptography* 9 (2019), pp. 2139–2164.

[123] 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: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel, 2013.

[124] R. C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *CRYPTO’87*. 1988, pp. 369–378.

[125] S. Micali and A. Shamir. “An Improvement of the Fiat-Shamir Identification and Signature Scheme”. In: *CRYPTO’88*. 1990, pp. 244–247.

[126] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. “Sprites and State Channels: Payment Networks that Go Faster Than Lightning”. In: *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. 2019, pp. 508–526.

[127] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. “Sprites and State Channels: Payment Networks that Go Faster Than Lightning”. In: *FC 2019*. 2019, pp. 508–526.

[128] A. Mirzaei, A. Sakzad, J. Yu, and R. Steinfeld. “FPPW: A Fair and Privacy Preserving Watchtower for Bitcoin”. In: *Financial Cryptography and Data Security*. Berlin, Heidelberg, 2021, pp. 151–169.

[129] R. Mitra. *Plasma Breakthrough: OmiseGO (OMG) announces the launch of Ari*. https://tinyurl.com/y262ttos. Accessed on: 2.1.2023.

[130] *Monero*. https://www.getmonero.org/resources/research-lab/. Accessed on: 2.1.2023.

[131] *More Viable Plasma*. https://tinyurl.com/y2gpmwov. 2018. Accessed on: 2.1.2023.

[132] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008).

[133] J. Nick, T. Ruffing, and Y. Seurin. “MuSig2: Simple Two-Round Schnorr Multi-signatures”. In: *CRYPTO 2021, Part I*. Virtual Event, 2021, pp. 189–221.

[134] K. Ohta and T. Okamoto. "A Digital Multisignature Scheme Based on the Fiat-Shamir Scheme". In: *ASIACRYPT'91*. 1993, pp. 139–148.

[135] T. Okamoto. "A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems". In: *ACM Trans. Comput. Syst.* 4 (1988), 432–441. ISSN: 0734-2071.

[136] T. Okamoto. "Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes". In: *CRYPTO'92*. 1993, pp. 31–53.

[137] *omiseGO*. https://github.com/omgnetwork/plasma-mvp. Accessed on: 2.1.2023.

[138] H. Ong and C.-P. Schnorr. "Fast Signature Generation With a Fiat-Shamir-Like Scheme". In: *EUROCRYPT'90*. 1991, pp. 432–440.

[139] Optimism. *Optimistic rollup overview*. https://github.com/ethereum-optimism/optimistic-specs/blob/0e9673af0f2cafd89ac7d6c0e5d8bed7c67b74ca/overview.md. Accessed on: 2.1.2023.

[140] R. Pass, E. Shi, and F. Tramèr. "Formal Abstractions for Attested Execution Secure Processors". In: *EUROCRYPT 2017, Part I*. 2017, pp. 260–289.

[141] A. K. Pedro Moreno-Sanchez. *Scriptless Scripts with ECDSA*. https://lists.linuxfoundation.org/pipermail/lightning-dev/2018-April/001221.html. 2018. Accessed on: 2.1.2023.

[142] *Plasma Debit*. https://tinyurl.com/yx936xzk. 2018. Accessed on: 2.1.2023.

[143] *Plasma EVM 2.0: state-enforceable construction*. https://tinyurl.com/2t953jc4. Accessed on: 2.1.2023.

[144] *Plasma snapp*. https://tinyurl.com/yxbza3pl. 2018. Accessed on: 2.1.2023.

[145] *Plasma World Map - the hitchhiker's guide to the plasma*. https://ethresear.ch/t/plasma-world-map-the-hitchhiker-s-guide-to-the-plasma/4333. 2018. Accessed on: 2.1.2023.

[146] *Plasmabits: Viable Stateful Sidechain*. https://tinyurl.com/4kemu8vk. Accessed on: 2.1.2023.

[147] A. Poelstra. *Lightning in Scriptless Scripts*. Mimblewimble team mailing list, https://lists.launchpad.net/mimblewimble/msg00086.html. Accessed on: 2.1.2023.

[148] A. Poelstra. *Scriptless scripts*. https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-03-mit-bitcoin-expo/slides.pdf. 2017. Accessed on: 2.1.2023.

[149] D. Pointcheval. "A New Identification Scheme Based on the Perceptrons Problem". In: *EUROCRYPT'95*. 1995, pp. 319–328.

[150] *PolyCrypt*. https://polycry.pt. Accessed on: 2.1.2023.

[151] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. http://plasma.io/plasma.pdf. 2017. Accessed on: 2.1.2023.

[152] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-chain Instant Payments*. https://lightning.network/lightning-network-paper.pdf. 2016. Accessed on: 2.1.2023.

[153] X. Qin, H. Cui, and T. H. Yuen. *Generic Adaptor Signature*. Cryptology ePrint Archive, Report 2021/161. https://eprint.iacr.org/2021/161. 2021.

[154] X. Qin et al. *BlindHub: Bitcoin-Compatible Privacy-Preserving Payment Channel Hubs Supporting Variable Amounts*. Cryptology ePrint Archive, Paper 2022/1735. https://eprint.iacr.org/2022/1735. 2022.

[155] *Raiden*. https://raiden.network. Accessed on: 2.1.2023.

[156] River. *What Is Taproot and How Does It Benefit Bitcoin?* https://river.com/learn/what-is-taproot/. Accessed on: 2.1.2023.

[157] B. Schmidt, S. Meier, C. J. F. Cremers, and D. A. Basin. “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties”. In: *CSF 2012 Computer Security Foundations Symposium*. 2012, pp. 78–94.

[158] C.-P. Schnorr. “Efficient Identification and Signatures for Smart Cards”. In: *CRYPTO’89*. 1990, pp. 239–252.

[159] C.-P. Schnorr. “Efficient Signature Generation by Smart Cards”. In: *Journal of Cryptology* 3 (1991), pp. 161–174.

[160] S.-T. Shen, A. Rezapour, and W.-G. Tzeng. “Unique Signature with Short Output from CDH Assumption”. In: *ProvSec 2015*. 2015, pp. 475–488.

[161] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. In: *NDSS 2017*. 2017.

[162] D. Siegel. *Understanding The DAO Attack*. https://tinyurl.com/2p8npu54. 2016. Accessed on: 2.1.2023.

[163] J. Spilman and M. Hearn. *Bitcoin contracts*. https://en.bitcoin.it/wiki/Contract. Accessed on: 2.1.2023.

[164] StarkWare. *Volition and the Emerging Data Availability spectrum*. https://medium.com/starkware/volition-and-the-emerging-data-availability-spectrum-87e8bfa09bb. Accessed on: 2.1.2023.

[165] *State channels/Counterfactual*. https://statechannels.org. Accessed on: 2.1.2023.

[166] J. Stern. “A New Identification Scheme Based on Syndrome Decoding”. In: *CRYPTO’93*. 1994, pp. 13–21.

[167] S.-F. Sun, M. H. Au, J. K. Liu, and T. H. Yuen. "RingCT 2.0: A Compact Accumulator-Based (Linkable Ring Signature) Protocol for Blockchain Cryptocurrency Monero". In: *ESORICS 2017, Part II.* 2017, pp. 456–474.

[168] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning". In: *2016 IEEE Symposium on Security and Privacy.* 2016, pp. 526–545.

[169] E. Tairi, P. Moreno-Sanchez, and M. Maffei. "A^2L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs". In: *2021 IEEE Symposium on Security and Privacy.* 2021, pp. 1834–1851.

[170] E. Tairi, P. Moreno-Sanchez, and M. Maffei. "Post-Quantum Adaptor Signature for Privacy-Preserving Off-Chain Payments". In: *Financial Cryptography and Data Security.* Berlin, Heidelberg, 2021, pp. 131–150.

[171] O. Team. *The oasis blockchain platform.* https://oasisprotocol.org/papers. Accessed on: 2.1.2023.

[172] S. A. K. Thyagarajan and G. Malavolta. "Lockable Signatures for Blockchains: Scriptless Scripts for All Signatures". In: *2021 IEEE Symposium on Security and Privacy.* 2021, pp. 937–954.

[173] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena. "Obscuro: A Bitcoin Mixer Using Trusted Execution Environments". In: *Proceedings of the 34th Annual Computer Security Applications Conference.* ACSAC '18. San Juan, PR, USA: Association for Computing Machinery, 2018, 692–701. ISBN: 9781450365697. DOI: 10.1145/3274694.3274750. URL: https://doi.org/10.1145/3274694.3274750.

[174] Trustnodes. *Ethereum Transactions Fall Off the Cliff, Three Plasma Projects Close to Release Says Buterin.* https://www.trustnodes.com/2018/07/05/ethereum-transactions-fall-off-cliff-three-plasma-projects-close-release-says-buterin. 2018. Accessed on: 2.1.2023.

[175] J. Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security 2018.* 2018, pp. 991–1008.

[176] 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.* 2019, pp. 41–61.

[177] B. Whitehat. *Roll up.* https://github.com/barryWhiteHat/roll_up. Accessed on: 2.1.2023.

[178] G. Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 2014 (2014), pp. 1–32.

[179] T. H. Yuen, S. Sun, J. K. Liu, M. H. Au, M. F. Esgin, Q. Zhang, and D. Gu. "RingCT 3.0 for Blockchain Confidential Transaction: Shorter Size and Stronger Security". In: *FC 2020*. 2020, pp. 464–483.

[180] M. Zamani, M. Movahedi, and M. Raykova. "RapidChain: Scaling Blockchain via Full Sharding". In: *ACM CCS 2018*. 2018, pp. 931–948.

[181] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *ACM CCS 2016*. 2016, pp. 270–282.

List of Figures

1.1. A PCN with three ledger channels α, β and λ. In this example, Alice sends 1 coin to Bob. 5
1.2. A virtual channel γ built over ledger channels α, β. 6

2.1. Two example transactions demonstrating our notations. 14
2.2. A Lightning style payment channel where Alice and Bob have x_A and x_B coins respectively. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ is the upper bound for the time needed to publish a transaction on a blockchain. 15
2.3. A generalized channel with outputs $((x_1, \varphi_1), \dots, (x_n, \varphi_n))$. pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain. 20
2.4. Funding of a virtual channel γ without validity. 25
2.5. The figure on the right shows the transactions published after the virtual channel is successfully offloaded. The figure on the left shows the transactions published after Alice successfully punishes Ingrid. The grayed transaction $\mathtt{TX}_\mathtt{s}^\mathtt{B}$ has not been posted on-chain. 25

3.1. Identification scheme protocol. 33
3.2. Digital signature schemes from identification schemes [106]. 33
3.3. Overview of our results. A full arrow represents a generic transformation, a dotted/dashed arrow represent a generic transformation that requires additional homomorphic/aggregation properties. . . . 35
3.4. Generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to an $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ scheme. 37
3.5. Two-party (pre-)signing protocol. In the pre-signing protocol, the combined randomness must be shifted as shown in the gray row. . . 39

4.1. The "Plasma World Map", illustrating the different flavors of Plasma protocols posted on Ethereum Research Forum [145]. 44
4.2. A Merkle tree storing 8 elements $x_0, \cdots, x_7$. 45

List of Tables

4.1. Plasma MVP vs Cash 54
4.2. Plasma vs PCHs 54

List of Abbreviations

PCNs	Payment Channel Networks
TEE	Trusted Execution Environment
UTXO	Unspent Transaction Output
PCHs	Payment Channels Hubs
UC	Universally Composability
SNARK	Succinct Non-Interactive Argument of Knowledge
PPT	Probabilistic Polynomial Time
DPT	Deterministic Polynomial Time
ID	Identification Scheme
KOSK	Knowledge of the Secret Key
PoW	Proof of Work

A. Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures

This chapter corresponds to our published article in Asiacrypt 2021 [11] (https://doi.org/10.1007/978-3-030-92075-3_22). The full version of this article can be found in [10].

Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures

Lukas Aumayr[1], Oguzhan Ersoy[2], Andreas Erwig[3], Sebastian Faust[3], Kristina Hostáková[4], Matteo Maffei[1], Pedro Moreno-Sanchez[5], and Siavash Riahi[3]

[1] Technische Universität Wien, Austria, first.lastname@tuwien.ac.at
[2] Delft University of Technology, Netherlands, o.ersoy@tudelft.nl
[3] Technische Universität Darmstadt, Germany, first.lastname@tu-darmstadt.de
[4] ETH Zürich, Switzerland, kristina.hostakova@inf.ethz.ch
[5] IMDEA Software Institute, pedro.moreno@imdea.org

Abstract. Decentralized and permissionless ledgers offer an inherently low transaction rate, as a result of their consensus protocol demanding the storage of each transaction on-chain. A prominent proposal to tackle this scalability issue is to utilize off-chain protocols, where parties only need to post a limited number of transactions on-chain. Existing solutions can roughly be categorized into: (i) application-specific channels (e.g., payment channels), offering strictly weaker functionality than the underlying blockchain; and (ii) state channels, supporting arbitrary smart contracts at the cost of being compatible only with the few blockchains having Turing-complete scripting languages (e.g., Ethereum).
In this work, we introduce and formalize the notion of *generalized channels* allowing users to perform any operation supported by the underlying blockchain in an off-chain manner. Generalized channels thus extend the functionality of payment channels and relax the definition of state channels. We present a concrete construction compatible with any blockchain supporting transaction authorization, time-locks and constant number of Boolean $\wedge$ and $\vee$ operations – requirements fulfilled by many (non-Turing-complete) blockchains including the popular Bitcoin. To this end, we leverage *adaptor signatures* – a cryptographic primitive already used in the cryptocurrency literature but formalized as a standalone primitive in this work for the first time. We formally prove the security of our generalized channel construction in the Universal Composability framework.
As an important practical contribution, our generalized channel construction outperforms the state-of-the-art payment channel construction, the Lightning Network, in efficiency. Concretely, it halves the off-chain communication complexity and reduces the on-chain footprint in case of disputes from linear to constant in the number of off-chain applications funded by the channel. Finally, we evaluate the practicality of our construction via a prototype implementation and discuss various applications including financially secured fair two-party computation.

Keywords: Blockchain, adaptor signatures, off-chain protocols and channels.

1 Introduction

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

Intuitively, off-chain protocols build a second layer over the blockchain (often referred to as the *layer-1*) by allowing the vast majority of transactions to be processed directly between the involved participants, with the blockchain being used only in

the initial setup and in case of disputes, thereby drastically improving transaction throughput and confirmation time.

While there exists a large variety of different off-chain (or layer-2) solutions (see, e.g., [6, 53, 30, 32] and many more), *payment channels* [10, 19, 47] are by far the most prominent one. Intuitively, a payment channel works in three phases. First, the two users *open* a channel by locking a certain amount of coins on-chain into an account controlled by both users. Then they perform an arbitrary amount of payments by exchanging authenticated messages *off-chain*. Finally, they *close* the channel by announcing the outcome of their trades to the ledger.

Off-chain computations in Ethereum. Ethereum supports on-chain transactions specified in a *Turing-complete scripting language*, which enables the execution of arbitrarily complex programs, also called smart contracts, thereby going beyond simple payments. The underlying blockchain is organized accordingly in the account-based model, in which the balance associated to an account is explicitly stored in its memory and programmatically updated via smart contracts. By leveraging the expressiveness of Turing-complete scripting languages, payment channels can be generalized into so-called *state channels* [43, 22, 23], whose functionality goes far beyond simple payments. Namely, state channels enable users to execute arbitrarily complex smart contracts in an off-chain manner, thereby making their execution faster and cheaper.

Turing-complete vs restricted scripting. The majority of current blockchains (e.g., Bitcoin, Zcash, Monero, and Cardano's ADA) only support a restricted scripting language and are based on the Unspent Transaction Output (UTXO) model: intuitively, they enable a restricted class of transactions, possibly conditioned to some events, that transfer money from an unspent transaction to a new unspent transaction. There are several reasons behind the choice of a limited scripting language. First, the simplicity of design and usage, which is believed to be beneficial for security: countless examples of smart contract vulnerabilities on Ethereum show that complex contract logic and increased expressiveness pave the way for critical bugs, which may have severe consequences for the stability of the underlying currency as shown by the infamous DAO hack [48]. Second, blockchains with simple transaction logic are less costly to maintain: this is important as transaction execution is done by many parties, and even normal users. Finally, restricted scripting languages are expressive enough to encode many interesting computations (e.g., lotteries [2], auctions [21], and more [8, 37, 7]).

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

1.1 Our contribution

Our main contribution is to put forward the notion of *generalized channels* – a generic extension of payment channels to support off-chain execution of *arbitrary transaction logic* supported by the underlying blockchain. State channels can hence be seen as a special case of generalized channels for blockchains with Turing-complete scripting languages. We briefly outline our main contributions below. A technical overview of our construction is given in Sec. 2.

Generalized channels. We show that if the underlying UTXO-based blockchain supports transaction authorization, time-locks and basic Boolean logic (constant number of $\wedge$, $\vee$ operations), then *any* transaction logic available on layer-1 can be lifted to layer-2 securely and generically.

As most cryptocurrencies, including the by far most prominent Bitcoin, satisfy the assumptions of our construction, they can benefit from generalized channels as a scalability solution. This, in particular, implies that our construction directly enables to execute *any* Bitcoin transaction off-chain. Moreover, we stress that our construction can also be deployed over any blockchain that can simulate a UTXO-based system, which, in particular, includes blockchains with support for Turing-complete smart contracts, e.g., Ethereum or Hyperledger Fabric [1].

A novel revocation mechanism for generalized channels. The main technical challenge in our generalized channel design is to propose an efficient mechanism for old channel state revocation while putting minimal assumptions on the scripting language of the underlying blockchain. The state-of-the-art approach, put forward by the Lightning Network [47], uses a punishment mechanism which allows the cheated party to claim all coins from the channel. As we argue, a straightforward generalization of the Ligthning-style revocation is unsuitable for generalized channels. Firstly, the blockchain communication complexity in case of misbehavior depends on the number of parallel conditional payments funded by the channel. This significantly increases the blockchain overhead when processing a punishment (if triggered). Secondly, the security of the revocation mechanism relies on state duplication, hence each off-chain transaction funded by the channel has to be performed twice (once on each duplicate). This is particularly problematic when channels are built on top of channels [26] as the off-chain communication complexity grows exponentially with the number of channel layers.

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

Formalization of adaptor signatures. A key idea of our novel revocation mechanism is to utilize an *adaptor signature scheme* [46] – a cryptographic primitive introduced by the cryptocurrency community to tie together the authorization of a transaction and the leakage of a secret value. Although adaptor signatures have been used in previous works (e.g. [41, 29, 45]), a formal definition has never been presented. We fill this gap by providing the first formalization of adaptor signatures and their security (in terms of cryptographic games), and proving that ECDSA and

Schnorr-based schemes satisfy our notions. We believe that our formalization and security analysis of adaptor signatures is of independent interest (see details on the impact of our work below).

Formalization of generalized channels. In order to formally define the security guarantees of a generalized channel protocol, we utilize the extended Universal Composability model allowing for global setup (the GUC model for short) put forward by Canetti et al. [15]. More precisely, we model money mechanics of an UTXO-based blockchain via a global ledger ideal functionality and provide an ideal specification of a generalized channel protocol via a novel ideal functionality. Thereafter, we prove that our generalized channel construction satisfies this ideal specification. The key challenges of our security analysis are to ensure the consistency of timings imposed by the blockchain processing delay, and to ensure that no honest party can ever lose coins by participating in a channel.

Evaluation and applications. We implemented our protocols and conducted an experimental evaluation, demonstrating how to use generalized channels as a building block for popular off-chain applications, like payment routing through a payment channel network (PCN) [47, 42, 41] and channel splitting [26]. Concretely, our evaluation demonstrates that, already when routing *one* payment through a channel, the amount of blockchain fees in case of a dispute is reduced by 28% compared to the state-of-the-art Lightning network solution. In practice, there have been cases of disputes in channels with 50 concurrent payments [40], which currently costs 553.66 USD in fees to resolve in Lightning and only 17.47 USD with generalized channels. For channel splitting, we reduce the transactions to be exchanged off-chain per sub-channel from exponential to constant.

Moreover, we discuss how to use generalized channels to realize the Claim-or-Refund functionality of Bentov and Kumaresan [8]. This functionality, can be used to build a fair two-party computation protocol over Bitcoin, where fairness is achieved by financially penalizing malicious parties. Realizing the Claim-or-Refund functionality, in particular, implies that generalized channels allow parties to execute any two-party computation off-chain.

1.2 Other Related Work

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

Off-chain protocols. As already mentioned before, there has been an extensive line of work on various types of payment channels [10, 19, 47] and payment channel networks (PCNs) [47, 42, 41]. However, these constructions only support simple payments and do not extend to support more complex transaction logic. The authors in [34] provide a formalization of the Lightning Network (LN) in the UC framework. This formalization is, however, tailored to the details of the current LN and cannot be leveraged to formalize generalized channels as we propose here. Most related to our work is the research on state channels [43, 22, 23], as these constructions allow to lift any transaction logic supported by the underlying blockchain off-chain. However,

state channels crucially rely on the underlying blockchain to support smart contracts and hence do not work for blockchains with restricted scripting language. Finally, eltoo [20] is a payment channel construction which does not rely on a punishment mechanism, yet requires Bitcoin to adapt a new scripting command (op-code). This op-code, however, has not been included to Bitcoin's scripting language in the past due to security concerns. In the case of address reuse or lazy wallet designs, funds can be stolen by replaying transactions [52]. Moreover, the security of the eltoo protocol has not been formally proven and it only supports simple payments.

Apart from payment and state channels, numerous other solutions have been proposed in order to perform heavy on-chain computation off-chain. For instance, various previous works (e.g., [18, 17, 35]) focus on realizing on-chain functionality off-chain by using Trusted Execution Environments which, however, inherently add an additional trust assumptions that may not hold in practice (e.g., [12, 16, 13]). A proposal to remove these assumptions is to use MPC protocols [8, 37], which however require collateral linear in the number of conditional payments. In contrast, generalized channels only require constant collateral for the execution of an arbitrary number of such payments. There have been proposals to remedy the collateral requirement in MPC protocols [9, 36, 38] but they are incompatible with many existing UTXO blockchains, including Bitcoin.[6]

Adaptor signatures. Poelstra [46] introduced the notion of adaptor signatures (AS), which intuitively allows to create partial signatures whose completion is conditioned on solving a cryptographic hard problem – a feature that has been proven useful in off-chain applications such as PCNs [41] and payment-channel hubs [49]. For instance, Malavolta et al. [41] use AS as building block to define and realize multi-hop payments in PCNs. Moreover, AS have been used as an off-the-shelf cryptographic building block for multi-path payments [25] and Monero-compatible PCNs [51]. Banasik et al. [5] construct a scheme satisfying a similar notion to AS in order to allow two parties to exchange a digital asset using cryptocurrencies that do not support Turing-complete programs. None of these works, however, define AS as a stand-alone primitive. Concurrently to our work, Fournier [29] attempts to formalize AS as an instance of one-time verifiable encrypted signatures [11]. Yet, the definition of [29] is weaker than the one we give in this work and does not suffice for the channel applications. Also concurrent to this work, Thyagarajan and Malavolta [50] define *lockable signatures.* While similar to AS in spirit, lockable signatures are a weaker primitive as the partial signature must be created honestly (e.g., through MPC) and the solution to the cryptographic hardness problem must be known beforehand. On the other hand, lockable signatures can be built from any signature scheme while AS cannot be constructed from unique signatures [27].

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

2 Background and Solution Overview

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

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

To ease the readability, we illustrate the transaction flows using charts (see Fig. 1 for examples). We depict transactions as rectangles with rounded corners. Doubled edge rectangles represent transactions published on the blockchain, while single edge rectangles are transactions that could be published on the blockchain, but they are not (yet). Transaction outputs are depicted as a box inside the transaction. The value of the output is written inside the output box and the output condition is written above the arrow coming from the output.

Conditions of transaction outputs might be fairly complex and hence it would be cumbersome to spell them out above the arrows. Instead, for frequently used conditions, we define the following abbreviated notation. If the output script contains (among other conditions) signature verification w.r.t. some public keys $pk_1, \dots, pk_n$ on the body of the spending transaction, we write all the public keys *below* the arrow and the remaining conditions *above* the arrow. Hence, information below the arrow denotes "who *owns* the output" and information above denotes "additional spending conditions". If the output script contains a check of whether a given witness hashes to a predefined h, we express this by writing the hash value h *above* the arrow. Moreover, if the output script contains a relative time-lock, i.e., a condition that is satisfied if and only if at least t rounds passed since the transaction was published, we write "$+t$" *above* the arrow. Finally, if the output script φ can be parsed as $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ for some $n \in \mathbb{N}$, we add a diamond shape to the corresponding transaction output. Each of the sub-conditions φ_i is then written above a separate arrow.

Payment channels. A payment channel [47] enables several payments between two users without submitting every single transaction to the blockchain. The cornerstone of payment channels is depositing coins into an output controlled by two users, who then authorize new deposit balances in a peer-to-peer fashion while having the guarantee that all coins are refunded at a mutually agreed time.

Fig. 1. (Left) tx is published on the blockchain. The output of value x_1 can be spent by a transaction containing a preimage of h and signed w.r.t. pk_A. The output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t rounds passed since tx was accepted by the blockchain. (Right) tx′ is not published yet. Its only output can be spent by a transaction whose witness satisfies $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

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

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

Lightning Network [47], the state-of-the-art payment channel for Bitcoin, implements such mechanism by introducing *two* commit transactions, denoted $\mathtt{TX_c^A}$ and $\mathtt{TX_c^B}$, per channel update, each of which contains a punishment mechanism for one of the users. In more detail (see also Fig. 2), the output of $\mathtt{TX_c^A}$ representing Alice's balance in the channel has a special condition. Namely, it can be spent by Bob if he presents a preimage of a hash value h_A or by Alice if certain number of rounds passed since the transaction was published. During a channel update, Alice chooses a value r_A, called the *revocation secret*, and presents the hash $h_A := \mathcal{H}(r_A)$ to Bob. Knowing h_A, Bob can create and sign the commit transaction $\mathtt{TX_c^A}$ with the built-in punishment for Alice (analogously for Bob and $\mathtt{TX_c^B}$). During the next channel update, parties first commit to the new state by creating and signing $\overline{\mathtt{TX}}_\mathtt{c}^\mathtt{A}$ and $\overline{\mathtt{TX}}_\mathtt{c}^\mathtt{B}$, and then *revoke* the old state by sending the revocation secrets to each other thereby enabling the punishment mechanism. If a malicious Alice now publishes the old commit transaction $\mathtt{TX_c^A}$, Bob can spend both of its outputs and claim all coins locked in the channel.

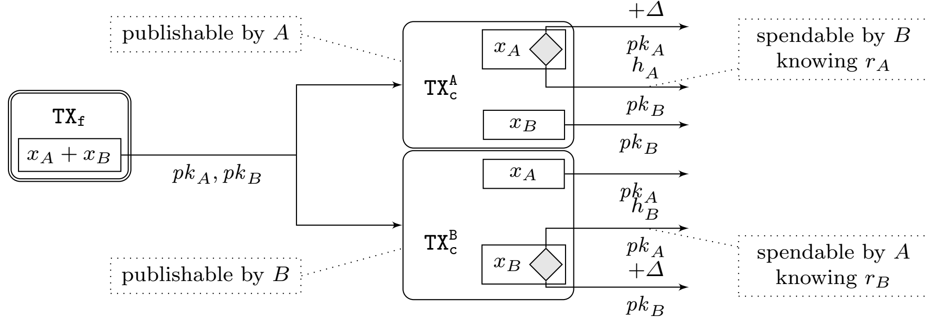

Fig. 2. A Lightning style payment channel where A has x_A coins and B has x_B coins. The values h_A and h_B correspond to the hash values of the revocation secrets r_A and r_B. Δ upper bounds the time needed to publish a transaction on a blockchain.

2.1 Solution Overview

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

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

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

One commit transaction per channel update. Another drawback of the Lightning-style revocation mechanism is the need for two commit transactions for the same channel state. While this is not an issue for simple payment channels, for generalized channels it might cause undesirable redundancy in terms of communication and computational costs. This comes from the fact that generalized channels support arbitrary output conditions and hence can be used as a source of funding for other off-chain applications, e.g., a fair two-party computation or another off-chain channel as we discuss later in this work (see Sec. 7). Such off-chain application would, however, have to "exist" twice. Once considering $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ being eventually published on-chain and once considering $\mathtt{TX}_\mathtt{c}^\mathtt{B}$. Especially when considering channels built on top of channels, the overhead grows exponentially. Our goal is to construct generalized channels that require only one commit transaction and hence avoid any redundancy.

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

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

To conclude, our solution, depicted in Fig. 3, requires only one commit transaction $\mathtt{TX}_\mathtt{c}$ per update. The commit transaction has one output that can be spent (i) by Alice if she knows Bob's revocation secret r_B *and* publishing secret y_B; (ii) by Bob if he knows Alice's revocation secret r_A *and* publishing secret y_A or (iii) by the split transaction $\mathtt{TX}_\mathtt{s}$ representing the channels state after some time. In the depicted construction, we assume that statement/witness pairs used for the adaptor signature scheme are public/secret keys of the blockchain signature scheme. Hence, testing if a party knows a publishing secret can be done by requiring a valid signature w.r.t. this public key. Let use emphasize that public/secret keys can also be used for the revocation mechanism instead of the hash/preimage pairs. This is actually preferable (not only in our construction but also in the Lightning-style channels) since the

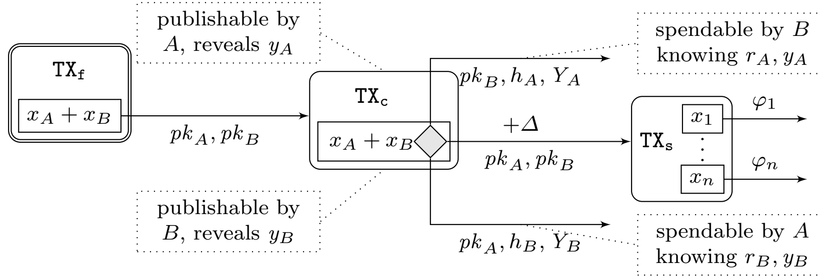

Fig. 3. A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. In the figure, pk_A denotes Alice's public key, (h_A, r_A) her revocation public/secret values, and (Y_A, y_A) her publishing public/secret values (analogously for Bob). The value of Δ upper bounds the time needed to publish a transaction on a blockchain.

punishment output script will only consist of signature verification, thereby requiring less complex scripting language. As a result, our solution does not only work over Bitcoin, but over any UTXO based blockchain that supports transaction authorization (if there exists an adaptor signature scheme w.r.t. the considered digital signature), relative time-locks and constant number of $\wedge$ and $\vee$ in output scripts.

3 Preliminaries

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

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

We next recall the definition of a hard relation R with statement/witness pairs (Y, y). Let L_R be the associated language defined as $\{Y \mid \exists y \text{ s.t. } (Y, y) \in R\}$. We say that R is a *hard relation* if the following holds: (i) There exists a PPT sampling

algorithm GenR that on input 1^n outputs a statement/witness pair $(Y, y) \in R$; (ii) The relation is poly-time decidable; (iii) For all PPT $\mathcal{A}$ the probability of $\mathcal{A}$ on input Y outputting a valid witness y is negligible.

Finally, we recall the definition of a non-interactive zero-knowledge proof of knowledge (NIZK) with online extractors as introduced in [28]. The online extractability property allows for extraction of a witness y for a statement Y from a proof π in the random oracle model and is useful for models where the rewinding proof technique is not allowed, such as UC. We need this property to prove our ECDSA-based adaptor signature scheme secure. More formally, a pair (P, V) of PPT algorithms is called a NIZK with an online extractor for a relation R, random oracle $\mathcal{H}$ and security parameter n if the following holds: (i) *Completeness*: For any $(Y, y) \in R$, it holds that $\mathsf{V}(Y, \mathsf{P}(Y, y)) = 1$ except with negligible probability; (ii) *Zero knowledge*: There exists a PPT simulator, which on input Y can simulate the proof π for any $(Y, y) \in R$. (iii) *Online Extractor*: There exist a PPT online extractor K with access to the sequence of queries to the random oracle and its answers, such that given (Y, π), the algorithm K can extract the witness y with $(Y, y) \in R$. An instance of such proof system is in [28].

4 Generalized channels

4.1 Notation and security model

To formally model the security of generalized channels, we use the global UC framework (GUC) [15] which extends the standard UC framework [14] by allowing for a global setup. Here we discuss our security model (which follows the previous works on off-chain channels [22, 23, 24]), only briefly and refer the reader to the full version of this paper [4] for more details.

We consider a protocol π that runs between parties from a fixed set $\mathcal{P} = \{P_1, \ldots, P_n\}$. A protocol is executed in the presence of an *adversary* $\mathcal{A}$ who can *corrupt* any party P_i at the beginning of the protocol execution (so-called static corruption). Parties and the adversary $\mathcal{A}$ receive their inputs from a special entity – called the *environment* $\mathcal{Z}$ – which represents anything "external" to the current protocol execution. We assume a synchronous communication network meaning that protocol execution happens in rounds, formalized via a global ideal functionality $\mathcal{F}_{clock}$ representing "the clock" [33]. Parties in the protocol are connected with authenticated communication channels with guaranteed delivery of exactly one round, formalized via an ideal functionality $\mathcal{F}_{GDC}$. For simplicity, we assume that all other communication (e.g., messages sent between the adversary and the environment) as well as local computation take zero rounds. Monetary transactions are handled by a global ideal ledger functionality $\mathcal{L}(\Delta, \Sigma, \mathcal{V})$, where Δ is an upper bound on the blockchain delay (number of rounds it takes to publish a transaction), Σ defines the signature scheme and $\mathcal{V}$ defines valid output conditions. Furthermore, the global ledger maintains a PKI.

Generalized channel syntax. A *generalized channel* γ is an attribute tuple $(\gamma.\mathsf{id}, \gamma.\mathsf{users}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the channel identifier, $\gamma.\mathsf{users} \in \mathcal{P} \times$

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

4.2 Ideal Functionality

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

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

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

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

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

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

To keep $\mathcal{F}$ generic, we parameterized it by two values T and k – both of which must be independent of the blockchain delay Δ. At a high level, the value T upper bounds the maximal number of consecutive off-chain communication rounds between channel users. Since different parts of the protocol might require different amount of communication rounds, the upper bound T might not be reached in all steps. For instance, channel creation might require more communication rounds than old state revocation. To this end, we give the power to the simulator to "speed-up" the process when possible. The parameter k defines the number of ways the channel state $\gamma.\mathsf{st}$ can be published on the ledger. As discussed in Sec. 2, in this work we

present a protocol realizing the functionality for $k = 1$ (see Fig. 3). A generalized channel construction using Lightning style revocation mechanism (see Fig. 2) would be a candidate protocol for $k = 2$.

We assume that the functionality maintains a set Γ of created channels in their latest state and the corresponding funding transaction tx. We present $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$ formally in Fig. 4. Here we discuss each part of the functionality at a high level and argue why it captures the aforementioned security and efficiency properties identified above. We abbreviate $\mathcal{F} := \mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(T,k)$.

Create. If $\mathcal{F}$ receives a message of the form $(\mathtt{CREATE}, \gamma, tid_P)$ from both parties in $\gamma.\mathsf{users}$ within T rounds, it expects a channel funding transaction to appear on the ledger $\mathcal{L}$ within Δ rounds. Such a transaction must spend both funding sources (defined by transaction identifiers tid_P, tid_Q) and contain one output of the value $\gamma.\mathsf{cash}$. If this is true, $\mathcal{F}$ stores this transaction together with the channel γ in Γ and informs both parties about the successful channel creation via the message CREATED (how this can be done within the UC model is discussed in the full version of this paper [4]). Since a CREATE message is required from both parties and both parties receive CREATED, "consensus on creation" holds.

Close. Any of the two parties can request closure of the channel via the message $(\mathtt{CLOSE}, id)$, where id identifies the channel to be closed. In case both parties request closure within T rounds, *peaceful closure* is expected. This means that a transaction, spending the channel funding transaction and whose output corresponds to the latest channel state $\gamma.\mathsf{st}$, should appear on $\mathcal{L}$ within Δ rounds. If only one of the parties requests closing, $\mathcal{F}$ executes the ForceClose subprocedure in which case such transaction is supposed to appear on $\mathcal{L}$ within 3Δ rounds modelling possible dispute resolution. In both cases, if the funding transaction is not spent before a certain round, an ERROR is returned to both users.

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

The prepare phase starts when $\mathcal{F}$ receives a vector of transaction identifiers $\vec{tid} = (tid_1, \dots, tid_k)$ from $\mathcal{S}$.[7] In the optimistic case, it is completed within $3T + t_{\mathsf{stp}}$ rounds and ends when the initiating party P receives an UPDATE–OK message from $\mathcal{F}$. The setup phase can be aborted by both the initiating party P and the other

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

party Q. This is achieved by P not sending the SETUP–OK and by Q not sending the UPDATE–OK message, respectively. This models two things. Firstly, the fact that Q might not agree with the proposed update and secondly, that setting up off-chain objects might fail in which case parties want to abort the channel update. The abort may also result in a forceful closing of the channel via the subprocedure ForceClose. It happens when one of the parties has sufficient information to enforce the new state on-chain, while the other does not.

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

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

Punish. In order to guarantee "instant finality with punishments", parties continuously monitor the ledger and apply the punishment mechanism if misbehavior is detected. This is captured by the functionality in the part "Punish" which is executed at the end of each round. The functionality checks if a funding transaction of some channel was spent. If yes, then it expects one of the following to happen: (i) a punish transaction appears on $\mathcal{L}$ within Δ rounds, assigning $\gamma.\mathsf{cash}$ coins to the honest party $P \in \gamma.\mathsf{users}$; or (ii) a transaction whose output corresponds to the latest channel state $\gamma.\mathsf{st}$ appears on $\mathcal{L}$ within 2Δ rounds, meaning that the channel is peacefully or forcefully closed. If none of the above is true, ERROR is returned. Hence, under the condition that no ERROR was returned, the security property "instant finality with punish" is satisfied.

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

Notation used in the formal description in Fig. 4. Messages sent between parties and $\mathcal{F}$ have the following format: (MESSAGE_TYPE, *parameters*). To shorten the description, we use following arrow notation: by $m \overset{t}{\hookrightarrow} P$, we mean "send the message m to party P in round t." and by $m \overset{t}{\hookleftarrow} P$, we mean "receive a message m from party P in round t". To indicate that a message should be sent/received before/after a certain round, we use inequality symbols above the arrows. When $\mathcal{F}$ expects $\mathcal{S}$ to set certain values (such as the vector of *tid*'s during the update process or the exact round in which a message should be sent to parties) and it does not do so, we implicitly assume that ERROR is returned. Since we do not aim to make any claims about privacy, we implicitly assume that every message that $\mathcal{F}$ receives/sends from/to a party is directly forwarded to $\mathcal{S}$. In the formal description, we treat the channel set Γ as a function which on input id outputs (X, tx), where X is a set of channels s.t. for every $\gamma \in X$ $\gamma.\mathsf{id} = id$, if such channel exists and $\perp$ otherwise. We denote

the script requiring signature of (only) P as $\texttt{One–Sig}_{pk_P}$. Moreover, we omit several natural checks that one would expect $\mathcal{F}$ to make. For example, messages with missing parameters should be ignored, channel instruction should be accepted only from channel users, etc. We formally define all checks as a functionality wrapper in the full version of this paper [4]. Finally, we omit the read queries that $\mathcal{F}$ sends to $\mathcal{L}$ in order to learn its state.

Upon $(\texttt{CREATE}, \gamma, tid_P) \overset{\tau_0}{\hookleftarrow} P$, distinguish:

Both agreed: If already received $(\texttt{CREATE}, \gamma, tid_Q) \overset{\tau}{\hookleftarrow} Q$, where $\tau_0 - \tau \leq T$: If tx s.t. $\mathsf{tx.In} = (tid_P, tid_Q)$ and $\mathsf{tx.Out} = (\gamma.\mathsf{cash}, \varphi)$, for some φ, appears on $\mathcal{L}$ in round $\tau_1 \leq \tau + \Delta + T$, set $\Gamma(\gamma.\mathsf{id}) := (\{\gamma\}, \mathsf{tx})$ and $(\texttt{CREATED}, \gamma.\mathsf{id}) \overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{users}$. Else stop.

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

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

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

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

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

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

At the end of every round τ_0: For each $id \in \{0,1\}^*$ s.t. $(X, \mathsf{tx}) := \Gamma(id) \neq \bot$, check if $\mathcal{L}$ contains tx' with $\mathsf{tx}'.\mathsf{In} = \mathsf{tx.txid}$. If yes, then define $S := \{\gamma.\mathsf{st} \mid \gamma \in X\}$, $\tau := \tau_0 + 2\Delta$ and distinguish: **Close:** If tx'' s.t. $\mathsf{tx}''.\mathsf{In} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{Out} \in S$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, set $\Gamma(id) := \bot$ and $(\texttt{CLOSED}, id) \overset{\tau_1}{\hookrightarrow} \gamma.\mathsf{users}$ if not sent yet.

Punish: If tx'' s.t. $\mathsf{tx}''.\mathsf{In} = \mathsf{tx}'.\mathsf{txid}$ and $\mathsf{tx}''.\mathsf{Out} = (\gamma.\mathsf{cash}, \texttt{One–Sig}_{pk_P})$ appears on $\mathcal{L}$ in round $\tau_1 \leq \tau$, for P honest, set $\Gamma(id) := \bot$, $(\texttt{PUNISHED}, id) \overset{\tau_1}{\hookrightarrow} P$ and stop.

Error: Else $(\texttt{ERROR}) \overset{\tau}{\hookrightarrow} \gamma.\mathsf{users}$.

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

Fig. 4. The ideal functionality $\mathcal{F}^{\mathcal{L}(\Delta, \Sigma, \mathcal{V})}(T, k)$. We abbreviate $Q := \gamma.\mathsf{otherParty}(P)$.

5 Adaptor Signatures

Our goal is to realize the ideal functionality for generalized channel for $k = 1$, meaning that there is only one way to publish the channel state on-chain. As explained at a high level in Sec. 2.1, we achieve our goal by utilizing an adaptor signature scheme – a cryptographic primitive that we discuss in this section.

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

Despite the fact that adaptor signatures have been used in previous works (e.g. [41] [29] [45]), none of these works has given a formal definition of the adaptor signature primitive and its security. In the following, we fill this gap and provide the first game-based formalization of adaptor signatures. As already mentioned, Erwig et al. [27] recently extended our definition to a two-party case.

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

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

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

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

Definition 2 (Existential unforgeability). *An adaptor signature scheme $\Xi_{R,\Sigma}$ is* aEUF–CMA *secure if for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ there exists a negligible function ν such that:* $\Pr[\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}(n) = 1] \leq \nu(n)$*, where the experiment* $\mathsf{aSigForge}_{\mathcal{A},\Xi_{R,\Sigma}}$ *is defined as follows:*

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

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

The second property, called *pre-signature adaptability*, protects the verifier. It guarantees that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be completed into a valid signature using a witness y with $(Y, y) \in R$. Notice that this property is stronger than the pre-signature correctness property from Def. 1, since we require that even pre-signatures that were not produced by pSign but are valid, can be completed into valid signatures.

Definition 3 (Pre-signature adaptability). *An adaptor signature scheme $\Xi_{R,\Sigma}$ satisfies pre-signature adaptability if for any message $m \in \{0,1\}^*$, any statement, witness pair $(Y, y) \in R$, any public key pk and any pre-signature $\tilde{\sigma} \in \{0,1\}^*$ with* $\mathsf{pVrfy}_{pk}(m, Y; \tilde{\sigma}) = 1$*, we have* $\mathsf{Vrfy}_{pk}(m; \mathsf{Adapt}(\tilde{\sigma}, y)) = 1$.

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

Definition 4 (Witness extractability). *An adaptor signature scheme $\Xi_{R,\Sigma}$ is* witness extractable *if for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function ν such that the following holds:* $\Pr[\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}(n) = 1] \leq \nu(n)$*, where the experiment* $\mathsf{aWitExt}_{\mathcal{A},\Xi_{R,\Sigma}}$ *is defined as follows*

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

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

Definition 5. *An adaptor signature scheme $\Xi_{R,\Sigma}$ is secure, if it is* aEUF–CMA *secure, pre-signature adaptable and witness extractable.*

Note that none of the security definitions explicitly states that pre-signatures are unforgeable. However, it is implied by the definitions as we discuss in the full version of this paper [4].

5.1 ECDSA-based Adaptor Signature

We now construct a provably secure adaptor signature scheme based on ECDSA digital signatures that are commonly used by blockchains. The construction presented here is similar to the construction put forward by [45], however some modifications are needed for the security proof. In addition to the ECDSA-based adaptor signature scheme presented here, we show a scheme based on Schnorr digital signatures, including correctness and security proofs, in the full version of this paper [4].

Recall the ECDSA signature scheme $\Sigma_{\mathsf{ECDSA}} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ for a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q. The key generation algorithm samples $x \leftarrow_\$ \mathbb{Z}_q$ and outputs $g^x \in \mathbb{G}$ as the public key and x as the secret key. The signing algorithm on input a message $m \in \{0,1\}^*$, samples $k \leftarrow_\$ \mathbb{Z}_q$ and computes $r := f(g^k)$ and $s := k^{-1}(\mathcal{H}(m) + rx)$, where $\mathcal{H}\colon \{0,1\}^* \to \mathbb{Z}_q$ is a hash function modeled as a random oracle and $f\colon \mathbb{G} \to \mathbb{Z}_q$ (i.e., f is typically defined as the projection to the x-coordinate since in ECDSA the group $\mathbb{G}$ consists of elliptic curve points). The verification algorithm on input a message $m \in \{0,1\}^*$ and a signature (r, s) verifies that $f(g^{s^{-1}\mathcal{H}(m)} X^{s^{-1}r}) = r$. One of the properties of the ECDSA scheme is that if (r, s) is a valid signature for m, then so is $(r, -s)$. Consequently, Σ_{ECDSA} does not satisfy $\mathsf{SUF\text{–}CMA}$ security which we need in order to prove its security. In order to tackle this problem we build our adaptor signature from the *Positive ECDSA* scheme which guarantees that if (r, s) is a valid signature, then $|s| \leq (q-1)/2$. The positive ECDSA has already been used in other works such as [5, 39]. This slightly modified ECDSA scheme is not only assumed to be $\mathsf{SUF\text{–}CMA}$ but also prevents having two valid signatures for the same message after the signing process, which is useful in practice, e.g., for threshold signature schemes based on ECDSA. As the ECDSA verification accepts valid positive ECDSA signatures, these signatures can be used by any blockchain that uses ECDSA, e.g., Bitcoin.

The adaptor signature scheme in [45] is presented w.r.t. a relation $R_g \subseteq \mathbb{G} \times \mathbb{Z}_q$ defined as $R_g := \{(Y, y) \mid Y = g^y\}$. The main idea of the construction is that a pre-signature (r, s) for a statement Y is computed by embedding Y into the r-component while keeping the s-component unchanged. This embedding is rather involved, since the value s contains a product of k^{-1}, r and the secret key. More

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

Fig. 5. ECDSA-based adaptor signature scheme.

concretely, to compute the pre-signature for Y, the signer samples a random k and computes $K := Y^k$ and $\tilde{K} := g^k$. It then uses the first value to compute $r := f(K)$ and sets $s := k^{-1}(\mathcal{H}(m) + rx)$. To ensure that the signer uses the same value k in K and $\tilde{K}$, a zero-knowledge proof that $(\tilde{K}, K) \in L_Y := \{(\tilde{K}, K,) \mid \exists k \in \mathbb{Z}_q \text{ s.t. } g^k = \tilde{K} \wedge Y^k = K\}$ is attached to the pre-signature. We denote the prover of the NIZK as P_Y and the corresponding verifier as V_Y. The pre-signature adaptation is done by multiplying the value s with y^{-1}, where y is the corresponding witness for Y. This adjusts the randomness k used in s to ky, and hence matches with the r value.

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

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

It might seem that we did not make it any easier for the reduction to learn a witness needed for creating pre-signatures. However, we exploit the fact that we are in the ROM and the reduction answers adversary's random oracle queries. Upon receiving a statement $I_Y := (Y, \pi)$ for which it must produce a valid pre-signature, it uses the random oracle query table to extract a witness from the proof π. Knowing the witness y and a signature (r, s), the reduction can compute $(r, s \cdot y)$ and execute the simulator of the NIZK_Y to produce a consistency proof π. This concludes the protocol description and the main proof idea. We refer the reader to the full version of the paper [4] for the detailed proof of the following theorem.

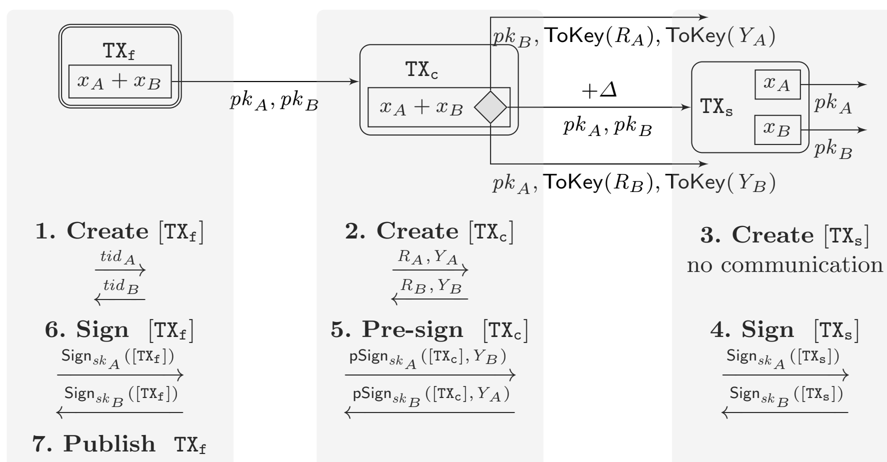

Fig. 6. Schematic description of the generalized channel creation protocol.

Theorem 1. *If the positive ECDSA signature scheme Σ_{ECDSA} is* SUF–CMA*-secure and R_g is a hard relation, $\Xi_{R'_g,\Sigma_{\mathsf{ECDSA}}}$ from Fig. 5 is a secure adaptor signature scheme in the ROM.*

6 Generalized Channel Construction

We now present a concrete protocol, denoted Π, that requires only one commit transaction, i.e., implements the punish-then-split mechanism. This is achieved by utilizing an adaptor signature scheme $\Xi_{R,\Sigma} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ for signature scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ used by the underlying ledger and a hard relation R. Throughout this section, we assume that statement/witness pairs of R are public/secret key of Σ. More precisely, we assume there exists a function ToKey that takes as input a statement $Y \in L_R$ and outputs a public key pk. The function is s.t. the distribution of $(\mathsf{ToKey}(Y), y)$, for $(Y, y) \leftarrow \mathsf{GenR}$, is equal to the distribution of $(pk, sk) \leftarrow \mathsf{Gen}$. We emphasize that both ECDSA and Schnorr based adaptor signatures satisfy this condition as discussed in the full version of the paper [4], where we also explain how to modify our protocol when this condition does not hold. Our protocol consists of four subprotocols: Create, Update, Close and Punish. We discuss each subprotocol separately at a high level here and refer the reader to the full version of the paper [4] for the pseudo-code description.

Channel creation. In order to create a channel γ, the users of the channel, say A and B, have to agree on the body of the funding transaction $[\mathtt{TX_f}]$, mutually commit to the first channel state defined by $\gamma.\mathsf{st} = ((x_A, \mathtt{One{-}Sig}_{pk_A}), (x_B, \mathtt{One{-}Sig}_{pk_B}))$, and sign and publish the funding transaction $\mathtt{TX_f}$ on the ledger. Recall that $\mathtt{One{-}Sig}_{pk}$ represents the script that verifies that the transaction is correctly signed w.r.t. the public key pk. Once $\mathtt{TX_f}$ is published, the channel creation is completed. Looking at Fig. 6, one can summarize the creation process as a step-by-step creation of transaction bodies from left to right, and then a step-by-step signature exchange on the transaction bodies from right to left. Let us elaborate on this in more detail.

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

Punish A: It is correctly signed w.r.t. $pk_B, \mathsf{ToKey}(Y_A), \mathsf{ToKey}(R_A)$;
Punish B: It is correctly signed w.r.t. $pk_A, \mathsf{ToKey}(Y_B), \mathsf{ToKey}(R_B)$;
Channel state: It is correctly signed w.r.t. pk_A and pk_B, and at least Δ rounds have passed since $\mathtt{TX_c}$ was published.

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

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

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

Channel closure. The purpose of the closing procedure is to collaboratively publish the latest channel state on the blockchain. The naive implementation is to let parties publish the latest agreed upon commit transaction and thereafter the

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

When parties want to close a channel, they first run a "final update". In short, the final update preserves the latest channel state, but removes the punishment layer. More precisely, parties agree on a new split transaction that has exactly the same outputs as the last split transaction but spends the funding transaction $\mathtt{TX_f}$ directly (i.e., **Steps 2+5** from Fig. 6 are skipped). Once parties jointly sign the split transaction, they can publish it on the ledger which completes the channel closure. If the final update fails, parties close the channel forcefully. Namely, they first publish the latest commit transaction, wait until the time for punishments expires. Then they post the split transaction representing the final channel state. It takes at most Δ rounds to publish the commit transaction and at most 2Δ rounds to publish the split transaction once the commit transaction is accepted which corresponds to the upper bound dictated by our ideal functionality. Since forceful closing might also be triggered during a channel update (as we discuss next), we define forceful closure as a separate subprocedure `ForceClose`.

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

Part (i) is very similar to the agreement on the initial commit and split transaction as described in the creation protocol (**Steps 2-5** in Fig. 6). There is one major difference coming from the fact that the new channel state $\vec{\theta}$ can contain outputs that fund other off-chain applications (such as sub-channels).[8] In order to set up these applications, the identifier of the new split transaction is needed. To this end, parties first prepare the commit (**Steps 2+3**) to learn the desired identifier and set up all applications off-chain. Once this is done, which is signaled by "`SETUP–OK`" and takes at most t_{stp} rounds, parties execute the second part of the committing phase (**Steps 4+5**).

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

We now discuss what happens if one party misbehaves during the update. As long as none of the parties pre-signed the new commit transaction, i.e., before **Step 5**, misbehavior simply implies update failure. A more problematic case is when the misbehavior occurs after at least one of the parties pre-signed the new commit transaction. This happens, e.g., when one party pre-signs the new commit but the other does not; or when one party revokes the old commit and the other does not. In each

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

of these situations, an honest party ends up in a hybrid state when the update is neither rejected nor accepted. In order to realize our ideal functionality requiring consensus on update in bounded number of rounds, our protocol instructs an honest party to `ForceClose` the channel. This means that the honest party posts the latest commit transaction that both parties agreed on to the ledger guaranteeing that $\mathtt{TX_f}$ is spent within Δ rounds. If the transaction spending $\mathtt{TX_f}$ is the new commit transaction, the channel is closed in the updated state. Otherwise, the update fails and either the channel is closed in the state before the update, or the punishment mechanism is activated and the honest party gets financially compensated (as discussed in the next paragraph).

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

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

Theorem 2. *Let Σ be a* $\mathsf{SUF\text{–}CMA}$ *secure signature scheme, R a hard relation and $\Xi_{R,\Sigma}$ a secure adaptor signature scheme. Let $\mathcal{L}(\Delta,\Sigma,\mathcal{V})$ be a ledger, where $\mathcal{V}$ allows for transaction authorization w.r.t. Σ, relative time-locks and constant number of Boolean operations $\wedge$ and $\vee$. Then the protocol Π UC-realizes the ideal functionality $\mathcal{F}^{\mathcal{L}(\Delta,\Sigma,\mathcal{V})}(3,1)$.*

The formal UC proof of the Theorem 2 can be found in the full version of this paper [4]. Let us here just argue at a high level, why our protocol satisfies the most complex property defined by the ideal functionality, i.e., instant finality with punishment.

We first argue that instant finality holds after the channel creation, meaning that each of the two parties (alone) is able to unlock her coins from a created channel if it was never updated. The pre-signature adaptability property of $\Xi_{R,\Sigma}$ guarantees that after a successful channel creation, each party P is able to adapt the pre-signature

of the other party Q on $[\mathtt{TX_c}]$ by using the publishing secret value y_P (corresponding to Y_P). Party P can now sign $[\mathtt{TX_c}]$ herself and post $\mathtt{TX_c}$ on the ledger. Since parties never signed any other transaction spending $\mathtt{TX_f}$, the posted $\mathtt{TX_c}$ will be accepted by the ledger within Δ rounds. Note that here we rely on the unforgeability of the signature scheme and the unforgeability of the adaptor signature scheme. Let us stress that parties have not revealed their revocation secrets, i.e, the values r_P and r_Q, to each other yet. Hardness of the relation R implies that none of the two parties is able to use the punishment mechanism of the published commit transaction. Thus, after Δ rounds, P can post the split transaction $\mathtt{TX_s}$ on the ledger by which she unlocks her x_P coins.

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

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

7 Applications

Our generalized channels support a variety of applications such as PCNs [47, 42, 41], payment channel hubs [49, 31], multi-path payments in PCNs [25], financially fair two-party computation [8], channel splitting [26], virtual payment channels [3] or watchtowers [44]. Furthermore, generalized channels prove to be highly versatile in interoperable applications, i.e., applications that run across multiple blockchains. As generalized channels rely only on on-chain signature verification, time-locked transactions and basic Boolean logic, they can be implemented on a multitude of different blockchains, easing thus the design and execution of cross-chain applica-

tions. Here, we first generally discuss which applications can be built on top of generalized channels and then focus on several concrete examples.

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

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

Fair two-party computation. One important example of an application that can be built on top of generalized channels is the *claim-or-refund* functionality introduced by Bentov and Kumaresan [8], and used in a series of work to realize multiple applications over Bitcoin [37]. At a high level, claim-or-refund allows one party, say A, to lock β coins that can be claimed by party B if she presents a witness satisfying a condition f. After a predefined number of rounds, say t, the payment of β coins is refunded back to A if the witness is not revealed.

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

Channel splitting. A generalized channel can be split into multiple sub-channels that can be updated independently in parallel. This idea appears already in [26] where two users A and B want to split a channel γ with coin distribution (α_A, α_B)

into two sub-channels γ_0 and γ_1 with the coin distributions (β_A, β_B) and $(\alpha_A - \beta_A, \alpha_B - \beta_B)$ respectively.

Executing multiple applications without prior channel splitting requires all applications to share a single funding source (i.e., that provided by the channel) and thus to be adjusted with every single channel update (i.e., even if the update is required for a single application), which might significantly increase the off-chain communication complexity. However, first splitting the channel into sub-channels effectively makes the execution of applications in each sub-channel independent of each other. For instance, two applications that benefit from channel splitting are *payment channels with watchtower* [44] and *virtual channels* [3] – both of which rely on generalized channels, and which we discuss next.

We elaborate on further applications in the full version of this paper [4].

8 Performance Analysis

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

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

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

In case of a dispute, the difference in on-chain cost is even more pronounced. To punish in LC, the honest party needs to spend $m + 1$ outputs: the one representing the balance of the malicious party and one per HTLC. This is in contrast to GC,

Table 1. Costs of lightning (LC) and generalized channels (GC) funding m HTLCs.

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

where the honest party publishes the punishment transaction only. As a result, the total size of on-chain transactions in the LC is $513 + m \cdot 410$ bytes, which cost around $13.52 + m \cdot 10.80$ USD. In GC, the on-chain transaction size is 663 bytes resulting in a cost of 17.47 USD. There have already been disputes for channels with 50 active HTLCs [40]. To settle such a dispute in LC, transactions with 21013 bytes or a cost of 553.66 USD have to be deployed. In GC, again we only need 663 bytes or 17.47 USD. GC thus reduce the on-chain cost from linear on m to constant in the case of a dispute as shown in Table 1.

Evaluation of channel splitting. The state duplication impacts other applications as well, e.g., channel splitting (see Sec. 7). For a LC, two commit transactions need to be exchanged per update. Hence, if we split a LC into two sub-channels, parties need to create these sub-channels for both commit transactions. Moreover, for each sub-channel two commit transactions are required. This is a total of 4 commit transactions per sub-channel. GC needs only one commitment and one split transactions per sub-channel.

After a channel split, sub-channels are expected to behave as normal channels. If we want to split a LC sub-channel further, we would need eight commit transactions (two for each of the four commitments) per sub-channel. Observe, that for every recursive split of a channel, the amount of LC commit transactions for the new subchannel doubles. For the mth split, we need 2^{m+1} additional commit transactions in the LC setting. In the GC setting, there is no state duplication, therefore the amount of transactions per sub-channel is always one commit and one split transaction. We reduce the complexity for additional transactions on the mth split from exponential to constant.

Acknowledgments

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

References

[1] E. Androulaki et al. “Hyperledger fabric: a distributed operating system for permissioned blockchains”. In: *EuroSys*. 2018.

[2] M. Andrychowicz et al. “Secure multiparty computations on Bitcoin”. In: *Commun. ACM* 4 (2016).

[3] L. Aumayr et al. “Bitcoin-Compatible Virtual Channels”. In: *IEEE S&P*. 2021.

[4] L. Aumayr et al. *Generalized Channels from Limited Blockchain Scripts and Adaptor Signatures*. Cryptology ePrint Archive, Report 2020/476. https://ia.cr/2020/476. 2020.

[5] W. Banasik et al. “Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts”. In: *ESORICS*. 2016.

[6] S. Bano et al. “SoK: Consensus in the Age of Blockchains”. In: *ACM AFT*. 2019.

[7] M. Bartoletti and R. Zunino. “BitML: A Calculus for Bitcoin Smart Contracts”. In: *CCS*. 2018.

[8] I. Bentov and R. Kumaresan. “How to Use Bitcoin to Design Fair Protocols”. In: *CRYPTO 2014, Part II*. 2014.

[9] I. Bentov et al. “Instantaneous Decentralized Poker”. In: *ASIACRYPT*. 2017.

[10] *Bitcoin Wiki: Payment Channels*. https://tinyurl.com/y6msnk7u.

[11] D. Boneh et al. “Aggregate and Verifiably Encrypted Signatures from Bilinear Maps”. In: *EUROCRYPT 2003*. 2003.

[12] F. Brasser et al. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *11th USENIX Workshop on Offensive Technologies*. 2017.

[13] J. V. Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *USENIX*. 2018.

[14] R. Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *42nd FOCS*. 2001.

[15] R. Canetti et al. “Universally Composable Security with Global Setup”. In: *TCC 2007*. 2007.

[16] G. Chen et al. “Pectre Attacks: Leaking Enclave Secrets via Speculative Execution”. In: *IEEE Euro S&P*. 2018.

[17] R. Cheng et al. “Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts”. In: *IEEE EuroS&P*. 2019.

[18] P. Das et al. “FastKitten: Practical Smart Contracts on Bitcoin”. In: *USENIX 2019*.

[19] C. Decker and R. Wattenhofer. “A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels”. In: *Stabilization, Safety, and Security of Distributed Systems 2015*. 2015.

[20] C. Decker et al. *eltoo: A Simple Layer2 Protocol for Bitcoin*. https://blockstream.com/eltoo.pdf.

[21] D. Deuber et al. *Minting Mechanisms for Blockchain – or – Moving from Cryptoassets to Cryptocurrencies*. Cryptology ePrint Archive, Report 2018/1110. https://eprint.iacr.org/2018/1110. 2018.

[22] S. Dziembowski et al. “General State Channel Networks”. In: *ACM CCS 18*.

[23] S. Dziembowski et al. "Multi-party Virtual State Channels". In: *EURO-CRYPT 2019, Part I*. 2019.
[24] S. Dziembowski et al. "Perun: Virtual Payment Hubs over Cryptocurrencies". In: *IEEE S&P 2019*.
[25] L. Eckey et al. *Splitting Payments Locally While Routing Interdimensionally*. ePrint Archive. `https://eprint.iacr.org/2020/555`. 2020.
[26] C. Egger et al. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks". In: *ACM CCS 19*.
[27] A. Erwig et al. "Two-Party Adaptor Signatures From Identification Schemes". In: *PKC*. 2021.
[28] M. Fischlin. "Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors". In: *CRYPTO 2005*. 2005.
[29] L. Fournier. *One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures*. `https://tinyurl.com/y4qxopxp`. 2019.
[30] L. Gudgeon et al. "SoK: Off The Chain Transactions". In: *FC*. 2020.
[31] E. Heilman et al. "TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub". In: *NDSS*. 2017.
[32] M. Jourenko et al. *SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies*. Cryptology ePrint Archive, Report 2019/352. `https://eprint.iacr.org/2019/352`. 2019.
[33] J. Katz et al. "Universally Composable Synchronous Computation". In: *TCC 2013*. 2013.
[34] A. Kiayias and O. S. T. Litos. "A Composable Security Treatment of the Lightning Network". In: *IEEE CSF 2020*.
[35] A. Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *IEEE S&P*. 2016.
[36] R. Kumaresan and I. Bentov. "Amortizing Secure Computation with Penalties". In: *ACM CCS 16*. 2016.
[37] R. Kumaresan and I. Bentov. "How to Use Bitcoin to Incentivize Correct Computations". In: *ACM CCS 14*. 2014.
[38] R. Kumaresan et al. "How to Use Bitcoin to Play Decentralized Poker". In: *ACM CCS*. 2015.
[39] Y. Lindell. "Fast Secure Two-Party ECDSA Signing". In: *CRYPTO 2017, Part II*. 2017.
[40] *lnchannels*. `https://ln.bigsun.xyz/`. 2020.
[41] G. Malavolta et al. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *NDSS 2019*.
[42] G. Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: *ACM CCS 17*. 2017.
[43] A. Miller et al. "Sprites and State Channels: Payment Networks that Go Faster Than Lightning". In: *FC 2019*.
[44] A. Mirzaei et al. "FPPW: A Fair and Privacy Preserving Watchtower For Bitcoin". In: *FC*. 2021.

[45] P. Moreno-Sanchez and A. Kate. *Scriptless Scripts with ECDSA*. https://tinyurl.com/yxtjo47l.

[46] A. Poelstra. *Scriptless scripts*. https://tinyurl.com/ludcxyz. 2017.

[47] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-chain Instant Payments*. https://tinyurl.com/q54gnb4. 2016.

[48] D. Siegel. *Understanding The DAO Attack*. https://tinyurl.com/2bzxkn7a. 2016.

[49] E. Tairi et al. "A^{2}L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs". In: *IEEE S&P*. 2021.

[50] S. A. K. Thyagarajan and G. Malavolta. "Lockable Signatures for Blockchains: Scriptless Scripts for All Signatures". In: *IEEE S&P*. 2021.

[51] S. A. K. Thyagarajan et al. *PayMo: Payment Channels For Monero*. Cryptology ePrint Archive. https://eprint.iacr.org/2020/1441. 2020.

[52] *Transcripts from coredev.tech Amsterdam 2019 meeting on SIGHASH NOINPUT*. https://tinyurl.com/49ryfutr.

[53] G. Wang et al. "SoK: Sharding on Blockchain". In: *ACM AFT 2019*.

B. Bitcoin-Compatible Virtual Channels

This chapter corresponds to our published article in IEEE Symposium on Security and Privacy (SP), 2021 [12] (https://doi.org/10.1109/SP40001.2021.00097). Our full version can be found in [9].

Bitcoin-Compatible Virtual Channels

Lukas Aumayr[1], Oguzhan Ersoy[2], Andreas Erwig[3], Sebastian Faust[3], Kristina Hostáková[4], Matteo Maffei[1], Pedro Moreno-Sanchez[5], and Siavash Riahi[3]

[1] Technische Universität Wien, Austria, first.lastname@tuwien.ac.at
[2] Delft University of Technology, Netherlands, o.ersoy@tudelft.nl
[3] Technische Universität Darmstadt, Germany, first.lastname@tu-darmstadt.de
[4] ETH Zürich, Switzerland, kristina.hostakova@inf.ethz.ch
[5] IMDEA Software Institute, pedro.moreno@imdea.org

Abstract. Current permissionless cryptocurrencies such as Bitcoin suffer from a limited transaction rate and slow confirmation time, which hinders further adoption. Payment channels are one of the most promising solutions to address these problems, as they allow the parties of the channel to perform arbitrarily many payments in a peer-to-peer fashion while uploading only two transactions on the blockchain. This concept has been generalized into payment channel networks where a path of payment channels is used to settle the payment between two users that might not share a direct channel between them. However, this approach requires the active involvement of each user in the path, making the system less reliable (they might be offline), more expensive (they charge fees per payment), and slower (intermediaries need to be actively involved in the payment). To mitigate this issue, recent work has introduced the concept of virtual channels (IEEE S&P'19), which involve intermediaries only in the initial creation of a bridge between payer and payee, who can later on independently perform arbitrarily many off-chain transactions. Unfortunately, existing constructions are only available for Ethereum, as they rely on its account model and Turing-complete scripting language. The realization of virtual channels in other blockchain technologies with limited scripting capabilities, like Bitcoin, was so far considered an open challenge.

In this work, we present the first virtual channel protocols that are built on the UTXO-model and require a scripting language supporting only a digital signature scheme and a timelock functionality, being thus backward compatible with virtually every cryptocurrency, including Bitcoin. We formalize the security properties of virtual channels as an ideal functionality in the Universal Composability framework and prove that our protocol constitutes a secure realization thereof. We have prototyped and evaluated our protocol on the Bitcoin blockchain, demonstrating its efficiency: for n sequential payments, they require an off-chain exchange of 9+2n transactions or a total of 3524+695n bytes, with no on-chain footprint in the optimistic case. This is a substantial improvement compared to routing payments in a payment channel network, which requires 8n transactions with a total of 3026n bytes to be exchanged.

1 Introduction

Permissionless cryptocurrencies such as Bitcoin [22] have spurred increasing interest over the last years, putting forward a revolutionary, from both a technical and economical point of view, payment paradigm. Instead of relying on a central authority for transaction validation and accounting, Bitcoin relies on its core on a decentralized consensus protocol for these tasks. The consensus protocol establishes and maintains a distributed ledger that tracks every transaction, thereby enabling public verifiability. This approach, however, severely limits the transaction throughput and confirmation time, which in the case of Bitcoin is around ten transactions per second, and confirmation of an individual transaction can take up to 60 minutes. This

is in stark contrast to central payment providers that offer instantaneous transaction confirmation and support orders of magnitude higher transaction throughput. These scalability issues hinder permissionless cryptocurrencies such as Bitcoin from serving a growing base of payments.

Within other research efforts [15, 29, 4], payment channels [7] have emerged as one of the most promising scalability solutions. The most prominent example that is currently deployed over Bitcoin is the so-called Lightning network [24], which at the time of writing hosts deposits worth more than 60M USD. A payment channel enables an arbitrary number of payments between users while committing only two transactions onto the blockchain. In a bit more detail, a payment channel between Alice and Bob is first created by a single on-chain transaction that deposits Bitcoins into a multi-signature address controlled by both users. The parties additionally ensure that they can get their Bitcoins back at a mutually agreed expiration time. They can then pay to each other (possibly many times) by exchanging authenticated off-chain messages that represent an update of their share of coins in the multi-signature address. The payment channel is finally closed when a user submits the last authenticated distribution of Bitcoins to the blockchain (or after the channel has expired).

Interestingly, it is possible to leverage a path of opened payment channels from the sender to the receiver with enough capacity to settle their payments off-chain, thereby creating a payment channel network (PCN) [24, 19]. Assume that Alice wants to pay Bob, and they do not have a payment channel between each other but rather are connected through an intermediary user Ingrid. Upon a successful off-chain update of the payment channel between Alice and Ingrid, the latter would update her payment channel with Bob to make the overall transaction effective. The key challenge is how to perform the sequence of updates atomically in order to prevent Ingrid from stealing the money from Alice without paying Bob. The standard technique for constructing PCNs requires the intermediary (e.g., Ingrid in the example from above) to be actively involved in each payment. This has multiple disadvantages, including (i) making the system less reliable (e.g., Ingrid might have to go offline), (ii) increasing the latency of each payment, (iii) augmenting its costs since each intermediary charges a fee per transaction, and (iv) revealing possibly sensitive payment information to the intermediaries [23, 27, 18].

An alternative approach for connecting multiple payment channels was introduced by Dziembowski et al. [12]. They propose the concept of *virtual channels* – an off-chain protocol that enables direct off-chain transactions without the involvement of the intermediary. Following our running example, a virtual channel can be created between Alice and Bob using their individual payment channels with Ingrid. Ingrid must collaborate with Alice and Bob only to create such virtual channel, which can then be used by Alice and Bob to perform arbitrarily many off-chain payments without involving Ingrid. Virtual channels offer strong security guarantees: each user does not lose money even if the others collude. A salient application of virtual payment channels is so-called payment hubs [12]. Since establishing a payment channel requires a deposit and active monitoring, the number of channels a

user can establish is limited. With payment hubs [12], users have to establish just one payment channel with the hub and can then dynamically open and close virtual channels between each other on demand. Interestingly, since in a virtual channel the hub is not involved in the individual payments, even transactions worth fractions of cents can be carried out with low latency.

The design of secure virtual channels is very challenging since, as previously mentioned, it has to account for all possible compromise and collusion scenarios. For this purpose, existing virtual channel constructions [12] require smart contracts programmed over an expressive scripting language and the account model, as supported in Ethereum. This significantly simplifies the construction since the deposit of a channel, and its distribution between the end-points are stored in memory and can programmatically be updated. On the downside, however, these requirements currently limit the deployment of virtual channels to Ethereum.

It was an *open question* until now if virtual channels could be implemented at all in UTXO-based cryptocurrencies featuring only a limited scripting language, like Bitcoin and virtually all other permissionless cryptocurrencies. We believe that answering this question is important for several reasons. First, by limiting the trusted computing base (i.e., the scripting functionality supported by the underlying blockchain), we reduce the on-chain complexity of the virtual channel protocol. As bugs in smart contracts are manifold and notoriously hard to fix, our construction eliminates an additional attack vector by moving the complexity to the protocol level (rather than on-chain as in the construction from [12]). Second, investigating the minimal functionality that is required by the underlying ledger to support complex protocols is scientifically interesting. One may view this as a more general research direction of building a lambda calculus for off-chain protocols. Concretely, our construction shows that virtual channels can be built with stateless scripts, while earlier constructions required stateful on-chain computation. Finally, from a practical perspective, our construction can be integrated into the Lightning Network (the by far most prominent PCN), and thus our solution can offer the benefits of virtual payment channels/hubs to a broad user base.

1.1 Our contributions

In this work, we develop the *first* protocols for building virtual channel hubs over cryptocurrencies that support limited scripting functionality. Our construction requires only digital signatures and timelocks, which are ubiquitously available in cryptocurrencies and well characterized. We also provide a comprehensive formal analysis of our constructions and benchmarks of a prototype implementation. Concretely, our contributions are summarized below.

– We present the first protocols for virtual channel hubs that are built for the UTXO-model and require a scripting language supporting only digital signature verification and timelock functionality, being thus compatible with virtually every cryptocurrency, including Bitcoin. Since in the Lightning network currently only 10 supernodes are involved in more than 25% of all channels, our technique can be used to reduce the load on these nodes, and thereby help to reduce latency.

– We offer two constructions that differ on whether (i) the virtual channel is guaranteed to stay off-chain for an encoded validity period, or (ii) the intermediary Ingrid can decide to offload the virtual channel (i.e., convert it into a direct channel between Alice and Bob), thereby removing its involvement in it. These two variants support different business and functionality models, analogous to non-preemptible and preemptible virtual machines in the cloud setting, with Ingrid playing the role of the service provider.

– We formalize the security properties of virtual channels as an ideal functionality in the UC framework [8], and prove that our protocols constitute a secure realization thereof. Since our virtual channels are built in the UTXO-model, our ideal functionality and formalization significantly differs from earlier work [12].

– We evaluate our protocol over two different PCN constructions, the Lightning Network (LN) [24] and Generalized channels (GC) [3], which extend LN channels to support functionality other than one-to-one payments. We show that for virtual channels on top of GC, n sequential payment operations require an off-chain exchange of $9 + 2 \cdot n$ transactions or a total of $3524 + 695 \cdot n$ bytes, as compared to $8 \cdot n$ transactions or $3026 \cdot n$ bytes when Ingrid routes the payment actively through the PCN. This means a virtual channel is already cheaper if two or more sequential payments are performed. For virtual channels over LN, n transactions require an off-chain exchange of $6292 + 2824 \cdot n$ bytes, compared to $4776 \cdot n$ bytes when routed through an intermediary. We have interacted with the Bitcoin blockchain to store the required transactions, demonstrating the compatibility of our protocol.

To summarize, for the first time in Bitcoin, we enable off-chain payments between users connected by payment channels via a hub without requiring the continuous presence of any intermediary. Hence, our solution increases the reliability and, at the same time, reduces the latency and costs of Bitcoin PCNs.

2 Background

In this section, we first provide the notation and preliminaries on UTXO-based blockchains. We then overview the basics of payment and virtual channels, referring the reader to [1, 19, 20, 12] for further details. We finally discuss the main technical challenges one needs to overcome when constructing Bitcoin-compatible virtual channels.

2.1 UTXO-based blockchains

We adopt the notation for UTXO-based blockchains from [3], which we shortly review below.

Attribute tuples Let T be a tuple of values, which we call in the following *attributes*. Each attribute in T is identified by a unique keyword, e.g., attr and referred to as $T.\mathsf{attr}$.

Outputs and transactions We focus on blockchains based on the Unspent Transaction Output (UTXO) model, such as Bitcoin. In the UTXO model, coins are held in *outputs* of transactions. Formally, an output θ is an attribute tuple $(\theta.\mathsf{cash}, \theta.\varphi)$, where $\theta.\mathsf{cash}$ denotes the amount of coins associated with the output and $\theta.\varphi$ denotes the conditions that need to be satisfied in order to spend the output. The condition $\theta.\varphi$ can contain any set of operations (also called scripts) supported by the considered blockchain. We say that a user P controls or owns an output θ if $\theta.\varphi$ contains only a signature verification w.r.t. the public key of P.

In a nutshell, a *transaction* in the UTXO model, maps one or more existing outputs to a list of new outputs. The existing outputs are called *transaction inputs*. Formally, a transaction tx is an attribute tuple and consists of the following attributes $(\mathsf{tx.txid}, \mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.TimeLock}, \mathsf{tx.Witness})$. The attribute $\mathsf{tx.txid} \in \{0,1\}^*$ is called the identifier of the transaction. The identifier is calculated as $\mathsf{tx.txid} := \mathcal{H}([\mathsf{tx}])$, where $\mathcal{H}$ is a hash function which is modeled as a random oracle and $[\mathsf{tx}]$ is the *body of the transaction* defined as $[\mathsf{tx}] := (\mathsf{tx.Input}, \mathsf{tx.Output}, \mathsf{tx.TimeLock})$. The attribute $\mathsf{tx.Input}$ is a vector of strings which identify the inputs of tx. Similarly, the outputs of the transaction $\mathsf{tx.Output}$ is the vector of new outputs of the transaction tx. The attribute $\mathsf{tx.TimeLock} \in \mathbb{N} \cup \{0\}$ denotes the absolute time-lock of the transaction, which intuitively means that transaction tx will not be accepted by the blockchain before the round defined by $\mathsf{tx.TimeLock}$. The time-lock is by default set to 0, meaning that no time-lock is in place. Lastly, $\mathsf{tx.Witness} \in \{0,1\}^*$ called the transaction's witness, contains the witness of the transaction that is required to spend the transaction inputs.

We use charts in order to visualize the transaction flow in the rest of this work. We first explain the notation used in the charts and how they should be read. Transactions are shown using rectangles with rounded corners. Double edge rectangles are used to represent transactions that are already published on the blockchain. Single edge rectangles are transactions that could be published on the blockchain, but they are not yet. Each transaction contains one or more boxes (i.e., with squared corners) that represent the outputs of that transaction. The amount of coins allocated to each output is written inside the output box. In addition, the output condition is written on the arrow coming from the output.

In order to be concise, we use the following abbreviations for the frequently used conditions. Most outputs can only be spent by a transaction that is signed by a set of parties. In order to depict this condition, we write the public keys of all these parties *below* the arrow. We use the command `One−Sig` and `Multi−Sig` in the pseudocode. Other additional spending conditions are written *above* the arrow. The output script can have a relative time lock, i.e., a condition that is satisfied if and only if at least t rounds are passed since the transaction was published on the blockchain. We denote this output condition writing the string "$+t$" *above* the arrow (and `CheckRelative` in the pseudocode). In addition to relative time locks, an output can also have an absolute time lock, i.e., a condition that is satisfied only if t rounds elapsed since the blockchain was created and the first transaction was posted on it. We write the string "$> t$" *above* the arrow for this condition. Lastly, an output's spending

Fig. 1: (Left) Transaction tx is published on the blockchain. The output of value x_1 can be spent by a transaction signed w.r.t. pk_B after round t_2, and the output of value x_2 can be spent by a transaction signed w.r.t. pk_A and pk_B but only if at least t_3 rounds passed since tx was accepted by the blockchain. (Right) Transaction tx′ is not published on the ledger. Its only output, which is of value x, can be spent by a transaction whose witness satisfies the output condition $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

condition might be a disjunction of multiple conditions. In other words it can be written as $\varphi = \varphi_1 \vee \cdots \vee \varphi_n$ for some $n \in \mathbb{N}$ where φ is the output script. In this case, we add a diamond shape to the corresponding transaction output. Each of the subconditions φ_i is then written above a separate arrow. An example is given in Figure 1.

2.2 Payment channels

A payment channel enables arbitrarily many transactions between users while requiring only two on-chain transactions. The first step when creating a payment channel is to deposit coins into an output controlled by two users. Once the money is deposited, the users can authorize new balance updates in a peer-to-peer fashion while having the guarantee that all coins are refunded at a mutually agreed time. In a bit more detail, a payment channel has three operations: *open*, *update* and *close*. We necessarily keep the description short and refer to [15, 3] for further reading.

Open Assume that Alice and Bob want to create a payment channel with an initial deposit of x_A and x_B coins, respectively. For that, Alice and Bob agree on a *funding transaction* (that we denote by $\mathtt{TX_f}$) that sets as inputs two outputs controlled by Alice and Bob holding x_A and x_B coins respectively, and transfers them to an output controlled by both Alice and Bob. When $\mathtt{TX_f}$ is added to the blockchain, the payment channel is effectively open.

Update Assume now that Alice wants to pay $\alpha \leq x_A$ coins to Bob. For that, they create a new *commit transaction* $\mathtt{TX_c}$ representing the commitment from both users to the new balance of the channel. The commit transaction spends the output of $\mathtt{TX_f}$ into two new outputs: (i) one holding $x_A - \alpha$ coins controlled by Alice; and (ii) the other holding $x_B + \alpha$ coins controlled by Bob. Finally, parties exchange signatures on the commit transaction, which serve as valid witnesses for $\mathtt{TX_f}$. At this point, Alice (resp. Bob) could add $\mathtt{TX_c}$ to the blockchain. Instead, they keep it locally in their memory and overwrite it when they agree on another commitment transaction $\overline{\mathtt{TX}}_\mathtt{c}$ representing a newer balance of the channel. This, however, leads to the problem that there exist several commitment transactions that can possibly be added to the blockchain. Since all of them are spending the same output, only one can be

accepted by the blockchain. Since it is impossible to prevent a malicious user from publishing an outdated commit transaction, payment channels require a mechanism that punishes such malicious behavior. This mechanism is typically called *revocation* and enables that an honest user can take all the coins locked in the channel if the dishonest user publishes an outdated commitment transaction.

Close Assume finally that Alice and Bob no longer wish to use the channel. Then, they can collaboratively close the channel by submitting the last commitment transaction $\overline{\mathtt{TX}}_{\mathtt{c}}$ that they have agreed on to the blockchain. After it is accepted, the coins initially locked at the channel creation via $\mathtt{TX}_{\mathtt{f}}$ are redistributed to both users according to the last agreed balance. As aforementioned, if one of the users submits an outdated commitment transaction instead, the counterparty can punish the former through the revocation mechanism.

The Lightning Network [24] defines the state-of-the-art payment channel construction for Bitcoin.

2.3 Generalized channels

The recent work of Aumayr et al. [3] proposes the concept of *generalized channels*. Generalized channels improve and extend payment channels (see Figure 2 for details) in two ways. First, they extend the functionality of payment channels by offering off-chain execution of any script that is supported by the underlying ledger. Hence, one may view generalized channels as state channels for blockchains with restricted scripting functionality. Second, and more important for our work, generalized channels significantly improve the on-chain and off-chain communication complexity. More concretely, this efficiency improvement is achieved by introducing a so-called *split transaction* (that we denote as $\mathtt{TX}_{\mathtt{s}}$) along with a *punish-then-split* paradigm. In contrast to regular payment channels that require one revocation process per output in the commit transaction, the punish-then-split approach decouples the revocation process from the number of outputs in the commit transaction. This allows moving from revocation for each output to a single revocation for the entire channel. As shown in Figure 2, the commit transaction ($\mathtt{TX}_{\mathtt{c}}$) is only responsible for the punishment, while the split transaction ($\mathtt{TX}_{\mathtt{s}}$) holds the actual outputs of the channel.

The efficiency of generalized channels is further improved since they only require a single commit transaction per channel. This is in contrast to the payment channels used by Lightning, which require two distinct commit transactions for each channel user. We will discuss in Section 3.4 why the punish-then-split paradigm (and requiring only one commit transaction) is useful in order to improve the efficiency of our virtual channels for Bitcoin.

To simplify terminology, we will use the term *ledger channel* for all channels that are funded directly over the blockchain.

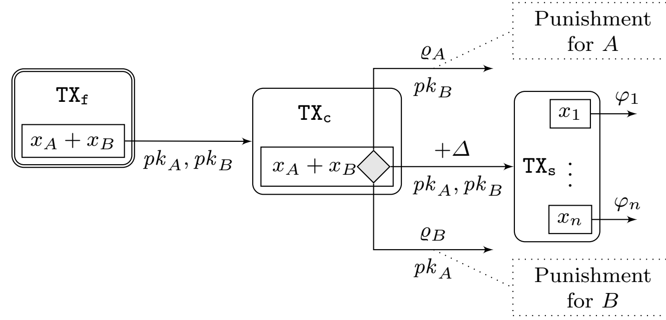

Fig. 2: A generalized channel in the state $((x_1, \varphi_1), \ldots, (x_n, \varphi_n))$. The value of Δ upper bounds the time needed to publish a transaction on a blockchain. The condition ϱ_A represents the verification of A' revocation secret and ϱ_B represents the verification of B' revocation secret.

2.4 Channel Networks

The aforementioned payment and generalized channels allow two parties to issue transactions between each other while having to communicate with the blockchain only during the creation and closure of the channel. This on-chain communication can further be reduced by using *channel networks.*

Payment Channel Networks (PCNs) A PCN is a protocol that allows parties to connect multiple ledger channels to form a payment channel network. In this network, a sender can route a payment to a receiver as long as both parties are connected by a path in the network. Suppose that Alice and Bob are not directly connected via a ledger channel, but instead both maintain a channel with an intermediary party (Ingrid). In a nutshell, Alice can pay Bob by sending her coins to Ingrid who then forwards them in her ledger channel to Bob. Importantly, the protocol must achieve atomicity, i.e., either both transfers from Alice to Ingrid and from Ingrid to Bob happen, or neither of them goes through. Current PCNs such as the Lightning network use the HTLC-technique (hash-time-lock transaction), which comes with several drawbacks as mentioned in the introduction: (i) *low reliability* because the success of payments relies on Ingrid being online; (ii) *high latency* as each payment must be routed through Ingrid; (iii) *high-cost* as Ingrid may charge a fee for each payment between Alice and Bob; and (iv) *low privacy* as Ingrid can observe each payment that happens between Alice and Bob. To mitigate these issues, virtual channels have been proposed.

Virtual Channels An alternative solution to connect two payment channels with each other is offered by the concept of *virtual channels* [12]. Virtual channels allow Alice and Bob to send payments between each other without the involvement of the intermediary Ingrid. In some sense, they thus mimic the functionality offered by ledger channels, with the difference that they are not created directly over the blockchain but instead over two ledger channels. More concretely, as shown in Figure 3, a virtual channel γ between Alice and Bob with intermediary Ingrid is constructed on

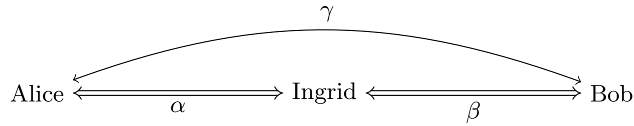

Fig. 3: A virtual channel γ built over ledger channels α, β.

top of two ledger channels α and β. Ingrid is required to participate in the initial creation and final closing of the virtual channel. But importantly, Ingrid is not involved in any balance updates that occur in the virtual channel. This overcomes the four drawbacks mentioned above. While these advantages over PCNs make virtual channels an attractive off-chain solution, their design is far from trivial. Previous work showed how to construct virtual channels over a ledger that supports Turing complete smart contracts [12, 13, 11]. The smart contract acts in the protocol as a trust anchor that parties can fall back to in case of malicious behavior. Through a rather complex protocol and careful smart contract design, existing virtual channel constructions guarantee that honest parties in the virtual channel will always get the coins they rightfully own. Unfortunately, most cryptocurrencies (including Bitcoin) do not offer Turing complete smart contracts, and hence the constructions from prior work cannot be used. In this work, we present a novel construction of virtual channels that makes only minimal assumptions on the underlying scripting functionality offered by the ledger.

3 Virtual Channels

In this section, we first give some notation before presenting the necessary properties for virtual channels and discussing design challenges. Finally, we present our protocol.

3.1 Definitions

We briefly recall some notation and definition for generalized channels [3] and extend the definition to generalized virtual channels. In order to make the distinction between the two types of channels clearer, we call the former generalized *ledger* channel (or ledger channels for short).

A *generalized ledger channel* as defined in [3] is a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st})$, where $\gamma.\mathsf{id} \in \{0,1\}^*$ is the identifier of the channel, $\gamma.\mathsf{Alice} \in \mathcal{P}, \gamma.\mathsf{Bob} \in \mathcal{P}$ are the identities of the parties using the channel, $\gamma.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ is a finite precision real number that represents the total amount of coins locked in this channel and $\gamma.\mathsf{st} = (\theta_1, \ldots, \theta_n)$ is the state of the channel. This state is composed of a list of *outputs*. Recall that each output θ_i has two attributes: the output value $\theta_i.\mathsf{cash} \in \mathbb{R}^{\geq 0}$ and the output condition $\theta_i.\varphi\colon \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \to \{0,1\}$. For convenience, we define a set $\gamma.\mathsf{endUsers} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}\}$ and a function $\gamma.\mathsf{otherParty}\colon \gamma.\mathsf{endUsers} \to \gamma.\mathsf{endUsers}$, which on input $\gamma.\mathsf{Alice}$ outputs $\gamma.\mathsf{Bob}$ and on input $\gamma.\mathsf{Bob}$ returns $\gamma.\mathsf{Alice}$.

A generalized *virtual channel* (or for short virtual channel) is defined as a tuple $\gamma := (\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st}, \gamma.\mathsf{Ingrid}, \gamma.\mathsf{subchan}, \gamma.\mathsf{fee}, \gamma.\mathsf{val})$. The attributes $\gamma.\mathsf{id}, \gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{cash}, \gamma.\mathsf{st}$ are defined as in the case of ledger channels. The additional attribute $\gamma.\mathsf{Ingrid} \in \mathcal{P}$ denotes the identity of the *intermediary* of the virtual channel γ. The set $\gamma.\mathsf{endUsers}$ and the function $\gamma.\mathsf{otherParty}$ are defined as before. Additionally, we also define the set $\gamma.\mathsf{users} := \{\gamma.\mathsf{Alice}, \gamma.\mathsf{Bob}, \gamma.\mathsf{Ingrid}\}$. The attribute $\gamma.\mathsf{subchan}$ is a function mapping $\gamma.\mathsf{endUsers}$ to a channel identifier; namely, the value $\gamma.\mathsf{subchan}(\gamma.\mathsf{Alice})$ refers to the identifier of the channel between $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Ingrid}$ (i.e., the id of α from the description above); similarly, the value $\gamma.\mathsf{subchan}(\gamma.\mathsf{Bob})$ refers to the identifier of the channel between $\gamma.\mathsf{Bob}$ and $\gamma.\mathsf{Ingrid}$ (i.e., β from the description above). The value $\gamma.\mathsf{fee} \in \mathbb{R}^{\geq 0}$ represents the fee charged by $\gamma.\mathsf{Ingrid}$ for her service of being an intermediary of γ. Finally, we introduce the attribute $\gamma.\mathsf{val} \in \mathbb{N} \cup \{\perp\}$. If $\gamma.\mathsf{val} \neq \perp$, then we call γ a *virtual channel with validity* and the value of $\gamma.\mathsf{val}$ represents the round number until which γ remains open. Channels with $\gamma.\mathsf{val} = \perp$ are called *virtual channels without validity.*

3.2 Security and efficiency goals

We briefly recall the properties of generalized channels as defined in [3] and state the additional properties that we require from virtual channels.

Security goals Generalized ledger channels must satisfy three security properties, namely (S1) Consensus on creation, (S2) Consensus on update and (S3) Instant finality with punish. Intuitively, properties (S1) and (S2) guarantee that successful creation of a new channel as well as successful update of an existing channel happens if and only if both parties agree on the respective action. Property (S3) states that if a channel γ is successfully updated to the state $\gamma.\mathsf{st}$ and $\gamma.\mathsf{st}$ is the last state that the channel is updated to, then an honest party $P \in \gamma.\mathsf{endUsers}$ can either enforce this state on the ledger or P can enforce a state where she gets all the coins locked in the channel. We say that a state st is *enforced* when a transaction with this state appears on the ledger.

Since virtual channels are generalized channels whose funding transaction is not posted on the ledger yet, the above stated properties should hold for virtual channels as well with two subtle but important differences: (i) the creation of a virtual channel involves three parties (Alice, Ingrid and Bob) and hence consensus on creation for virtual channels can only be fulfilled if all three parties agree on the creation; (ii) the finality (i.e., offloading) of the virtual channel depends on whether Alice is

	L-Security	V-Security			Efficiency		
	S1 – S3	V1	V2	V3	E1	E2	E3
L	✓	-	-	-	✗	✓	✗
VC-V	✓	✓	✗	✓	✓	✓	✓
VC-NV	✓	✓	✓	✗	✓	✓	✓

Table 1: Comparison of security and efficiency goals for ledger channels (L), virtual channels with validity (VC-V) and virtual channels without validity (VC-NV).

expected to offload the virtual channel within a predetermined validity period (virtual channel with validity VC-V) or the offload task is delegated to the intermediary Ingrid without having a predefined validity period (virtual channel without validity VC-NV). In order to account for these two differences, virtual channels should also satisfy the following properties:

(V1) Balance security: If γ is a virtual channel and $\gamma.\mathsf{Ingrid}$ is honest, she never loses coins, even if $\gamma.\mathsf{Alice}$ and $\gamma.\mathsf{Bob}$ collude.
(V2) Offload with punish: If γ is a virtual channel *without* validity (VC-NV), then $\gamma.\mathsf{Ingrid}$ can transform γ to a ledger channel. Party $P \in \gamma.\mathsf{endUsers}$ can initiate the transformation which either completes or P can get financially compensated.
(V3) Validity with punish: If γ is a virtual channel *with* validity (VC-V), then $\gamma.\mathsf{Alice}$ can transform γ to a ledger channel. If γ is not transformed into a ledger channel or closed before time $\gamma.\mathsf{val}$, $\gamma.\mathsf{Ingrid}$ and $\gamma.\mathsf{Bob}$ can get financially compensated.

We first note that the instant finality with punish property (S3) does not provide any guarantees for $\mathsf{Ingrid} \notin \gamma.\mathsf{endUsers}$, which is why we need to define (V1) for virtual channels. Properties (V2) and (V3) point out the main difference between VC-NV and VC-V. In a VC-NV γ, Ingrid is able to free her collateral from γ at any time by transforming the channel between Alice and Bob from a virtual channel to a ledger channel. Furthermore, in case Alice and Bob transform the virtual channel to a ledger channel or even misbehave, honest Ingrid is guaranteed that she will receive the collateral back. In a VC-V γ, Ingrid cannot transform a virtual channel into a ledger channel at any time she wants. Instead, there is a pre-agreed point in time, defined by $\gamma.\mathsf{val}$, until when $\gamma.\mathsf{endUsers}$ have to close the virtual channel or transform it into a ledger channel (Ingrid's collateral is freed in both cases). If $\gamma.\mathsf{endUsers}$ fail to do so, Ingrid can get her collateral back through a punishment mechanism. Hence, $\gamma.\mathsf{endUsers}$ have a guarantee that their VC-V will remain a virtual channel until a certain round, after which they must ensure its closure or transformation to avoid punishments.

Efficiency goals Lastly, we define the following efficiency goals, which describe the number of rounds certain protocol steps require:

(E1) Constant round creation: Successful creation of a virtual channel takes a constant number of rounds.
(E2) Optimistic update: For a channel γ, this property guarantees that in the optimistic case when both parties in $\gamma.\mathsf{endUsers}$ are honest, a channel update takes a constant number of rounds.
(E3) Optimistic closure: In the optimistic case when all parties in $\gamma.\mathsf{users}$ are honest, the closure of a virtual channel takes a constant number of rounds.

Let us stress that property (E2) is common for all off-chain channels (i.e., both ledger and virtual channels). The properties (E1) and (E3) capture the additional property

of virtual channels that in the optimistic case when all parties behave honestly, the entire life-cycle of the channel is performed completely off-chain.

We compare the security and efficiency goals for different types of channels in Table 1. We formalize these properties as a UC ideal functionality in Appendix A.

3.3 Design Challenges for Constructing Virtual Channels

The main challenges that arise when constructing Bitcoin-compatible virtual channels stem from the need to ensure the security properties (V1) - (V3) as presented in the previous section. Namely, to guarantee balance security to the intermediary, we need to ensure that the virtual channel creation and closure is reflected symmetrically and synchronously on both underlying ledger channels. We identify this as a challenge (C1). As we discuss in more detail below, this can be solved by giving the intermediary the right of a "last say" in the virtual channel creation and closure procedures. However, a malicious intermediary could abuse such power and block virtual channel closure indefinitely. Therefore, the second challenge (C2) is to design a punishment mechanism that allows virtual channel users to either enforce closure or claim financial compensation. We provide some further details below.

Synchronous create and close (C1) The creation and closure of a virtual channel are done by updating the underlying ledger channels. In order to guarantee balance security for the intermediary, we must ensure that updates on both ledger channels are symmetric and either both of them succeed or both of them fail. That is, if the intermediary Ingrid loses coins in one ledger channel as a result of the virtual channel construction, then she has the guarantee of gaining the same amount of coins from the other ledger channel. Such an atomicity property can be achieved by allowing Ingrid to be the reacting party in both ledger channel update procedures. Namely, Ingrid has to receive symmetric update requests from both Alice and Bob before she confirms either of them.

As a result, Ingrid has the power to block a virtual channel creation and closure. For a virtual channel creation, this is not a problem. It simply represents the fact that Ingrid does not want to be an intermediary, and hence Alice and Bob have to find a different party. However, for virtual channel closing, this power of the intermediary results in a violation of the instant finality property for Alice and Bob, and requires a more involved mechanism.

Enforcing virtual channel state (C2) In contrast to standard ledger channels that rely on funding transactions that are published on the ledger, the funding transactions of a virtual channel are, in the optimistic case (i.e., when parties are honest), kept off-chain. In case of misbehavior (e.g., when malicious Ingrid refuses to close the virtual channel), however, honest parties must be able to publish the virtual channel funding transaction to the blockchain in order to enforce the latest state of the virtual channel. Unfortunately, the funding transactions can only be published if *both* of the underlying channels are closed in a state which funds the virtual channel. The fact that the virtual channel participants, Alice and Bob, respectively

have control over just one of the underlying ledger channels further complicates this situation. For instance, one of the underlying ledger channels may be updated or closed maliciously at any time which would prevent the publishing of the funding transaction on the ledger.

3.4 Virtual Channel Protocol

We now show how to build virtual channels on top of generalized channels. We later discuss in Section 3.4 how our construction can be built over other channels such as Lightning and why generalized channels offer better efficiency.

As mentioned in the previous section, virtual channels are created and closed through an update of the underlying ledger channels. Hence, let us recall the update process of ledger channels, depicted as $\mathsf{UpdateChan}$ in Figures 4 and 5, before explaining our construction in more detail. The update procedure consists of 4 steps, namely (1) the *Initialization* step, during which parties agree on the new state of the channel, (2) the *Preparation* step, where parties generate the transactions with the given state, (3) the *Setup* during which parties exchange their application-dependent data (e.g., for building virtual channels), and finally (4) the *Completion* step where parties commit to the new state and revoke the old one. We refer the reader to [3] for more details.

High level protocol description We are now prepared to present a high-level description of our modular virtual channel protocol and explain how to solve the main technical challenges when designing virtual channels. In a nutshell, this modular protocol gives a generic framework on how to design virtual channels. Afterwards, we show how to instantiate this modular protocol with our virtual channel construction without validity. For the instantiation with our construction with validity, we refer the reader to Appendix B. We present the formal pseudocode for the modular protocol in Appendix C.

Create Let γ be a virtual channel that $A := \gamma.\mathsf{Alice}$ and $B := \gamma.\mathsf{Bob}$ want to create, using their generalized ledger channels with $I := \gamma.\mathsf{Ingrid}$. At a high level, the creation procedure of a virtual channel is a synchronous update of the underlying ledger channels. Given the ledger channels, we proceed as follows (see Figure 4).

As a first step, each party $P \in \{A, B\}$ initiates an update of the respective ledger channel with I (step ①) who, upon receiving both update requests, checks if the requested states (i.e., θ_A and θ_B) are consistent. The parties use the identifiers tid_A and tid_B of their subchannels in order to build the virtual channel (step ②). Next, all three parties engage in a setup phase, in which the structure of the virtual channel is built (step ③). More concretely, all three parties agree on a funding transaction of the virtual channel which when published on the blockchain transforms the virtual channel to a ledger channel. When the setup phase is completed, i.e., the virtual channel structure has been built, the parties complete the ledger channel update procedures (step ④). It is crucial for the intermediary I to have the role of a reacting party during both channel updates. This gives her the power to wait until she is sure

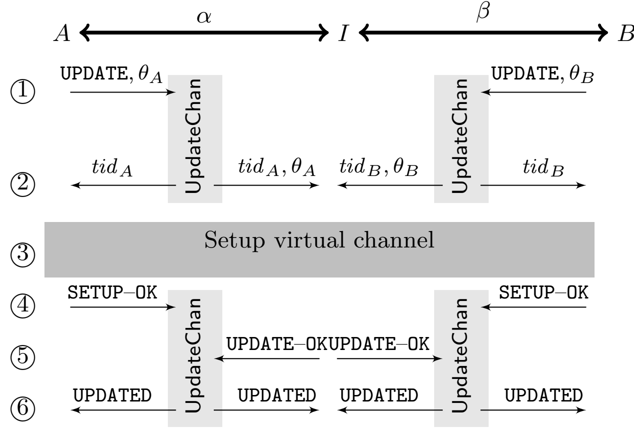

Fig. 4: Modular creation procedure of a virtual channel on top of two ledger channels α and β.

that both updates will complete successfully and only then give her the final update agreement (step ⑤). Upon a successful execution, parties consider the channels as updated (step ⑥), which implies that the virtual channel γ was successfully created.

Update Updating the virtual channel essentially works in the same way as the update procedure of a ledger channel. As long as the update is successful or peacefully rejected (meaning that the reacting party rejects the update), the parties act as instructed in the ledger channel protocol. The situation is more delicate when the update fails because one of the parties misbehaved and aborted the procedure.

We note that aborts during a channel update might cause a problematic asymmetry between the parties. For instance, when one party already signed the new state of the channel while the other one did not; or when one party already revoked the old state of the channel but the other one did not. In a standard ledger channel, these disputes are resolved by a force close procedure, meaning that the honest party publishes the latest valid state on the blockchain, thereby forcefully closing the channel. Hence, within a finite number of rounds, the dispute is resolved and the instant finality property is preserved. We apply a similar technique for virtual channels. The main difference is that a virtual channel is not funded on-chain. Hence, we first need to offload the virtual channel to the ledger. In other words, we first need to transform a virtual channel into a ledger channel by publishing its funding transaction on-chain. This process is discussed later in this section. Once the funding transaction is published, the dispute is handled in the same way as for ledger channels.

Close The closure of a virtual channel is done by updating the underlying ledger channels α and β according to the latest state of the virtual channel $\gamma.\mathsf{st}$. To this end, each party $P \in \{A, B\}$ computes the new state for the ledger channel $\vec{\theta}_P := \{(c_P, \texttt{One–Sig}_{pk_P}), (\gamma.\mathsf{cash} - c_P, \texttt{One–Sig}_{pk_I})\}$ where c_P is the latest balance of P in γ. All parties update their ledger channels according to this state.

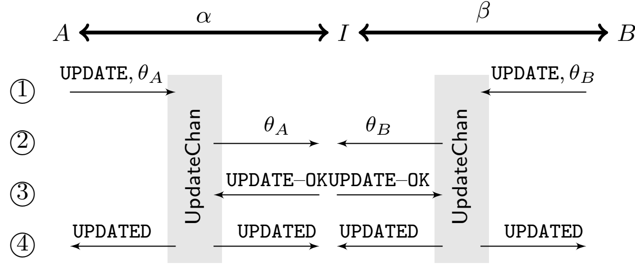

Fig. 5: Modular close procedure of a virtual channel on top of two ledger channels α and β. For $P \in \{A, B\}$, $\vec{\theta}_P := \{(c_P, \texttt{One-Sig}_{pk_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \texttt{One-Sig}_{pk_I})\}$ where $\gamma.\mathsf{st} = \Big((c_P, \texttt{One-Sig}_{pk_P}), (c_Q, \texttt{One-Sig}_{pk_Q})\Big)$.

In a bit more detail, the closing procedure of a virtual channel proceeds as follows (see Figure 5). Each party P initiates an update of the underlying ledger channel with state $\vec{\theta}_P$ (step ①). Since both ledger channels must be updated synchronously, I waits for both parties to initiate the update procedure. Upon receiving the states from both parties (step ②), I checks that the states are consistent and if so, she agrees to the update of both ledger channels (step ③). Finally, after all parties have successfully revoked the previous ledger channel state, the virtual channel is considered to be closed.

In the pessimistic case (if the states $\vec{\theta}_A$ and $\vec{\theta}_B$ are inconsistent, revocation fails or I remains idle), parties must forcefully close their virtual channel by publishing the funding transaction (offloading) and closing the resulting ledger channel. This, together with the fact that I plays the role of the reacting party in its interactions with A and B, addresses the challenge (C1) as mentioned in Section 3.3.

Offload During the offload procedure, parties try to publish the funding transaction of the virtual channel γ which effectively transforms the virtual channel into a ledger channel. In a nutshell, during this procedure, parties try to publish the commit and split transactions of both underlying ledger channels and afterward the funding transaction of the virtual channel. In case offloading is prevented by some form of malicious behavior, parties can engage in the punishment procedure to ensure that they do not lose any funds.

Punish The concept of punishment in virtual channels is similar to that in ledger channels; namely in case that the latest state of a channel cannot be posted on the ledger, honest A or B are compensated by receiving all coins of the virtual channel while honest I will not lose coins. If the funding transaction of the virtual channel is posted on the ledger, the virtual channel is transformed into a ledger channel and parties can execute the regular punishment protocol for ledger channels. In addition to the ledger channel's punishment procedure, parties can punish if the funding transaction of γ cannot be published. Since this punishment, however, differs for each concrete instantiation, we will explain it in more detail for our protocol without validity in the following section (and in Appendix B for the case with validity).

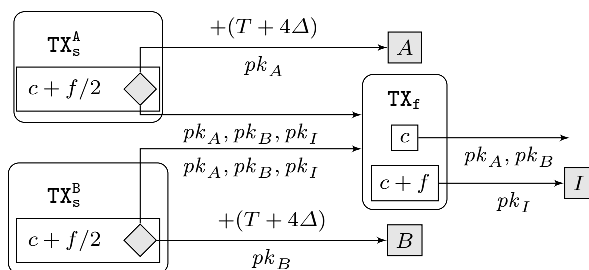

Fig. 6: Funding of a virtual channel γ without validity. T upper bounds the number of off-chain communication rounds between two parties for any operation in the ledger channel.

The offloading and punishment procedure together tackles challenge (C2) from Section 3.3.

Concrete Instantiation Without Validity We now describe how the modular protocol explained above can be concretely instantiated with our construction for virtual channels without validity.

Create In our construction without validity, A and B must "prepare" the virtual channel during the setup procedure (step ③ in create of the modular protocol). This is done by executing the creation procedure of a regular ledger channel, i.e., they create a funding transaction with inputs tid_A and tid_B, as well as a commit and split transactions that spend the funding transaction. Once all three transactions are created, A and B sign them and exchange their signatures. Note that this corresponds to a normal channel opening, with the mere difference that the funding transaction is not published to the blockchain. In order to complete the virtual channel setup, A and B send the signed funding transaction to I who, upon receiving both signatures, sends her own signature on the transaction back to A and B. At this stage, the virtual channel is prepared, however, the creation is not completed yet. In order to finish the creation procedure, A, I, and B have to finish the update of their respective ledger channels. Once this is done, the virtual channel has been successfully created.

We illustrate the transaction structure prepared during the creation process in Figure 6. The funding transaction of the virtual channel $\mathtt{TX_f}$, which is generated during the create procedure, takes as input coins from both, the ledger channel α (represented by $\mathtt{TX_s^A}$) and the ledger channel β (represented by $\mathtt{TX_s^B}$). Both ledger channels jointly contribute a total of $2c + f$ coins so that c coins are later used to setup the virtual channel and the remaining $c + f$ coins are I's collateral and the fees paid to I for providing the service for A and B.[6] I's collateral and fees in the funding transaction $\mathtt{TX_f}$ are the reason why I has to proactively monitor the virtual channel as she has an incentive to publish $\mathtt{TX_f}$ in case any party misbehaves.

[6] For simplicity we assume each of the parties contributes $f/2$ coins to I's total fees in addition to $c/2$ coins for funding the virtual channel.

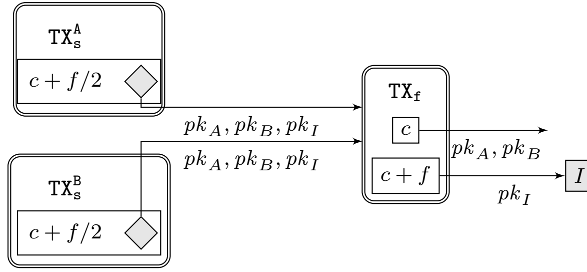

Fig. 7: Transactions published after a successful offload.

Offload I is always able to offload the virtual channel by herself (i.e., without having to cooperate with another party) which guarantees that I can redeem her collateral at any time. We note that $P \in \{A, B\}$ can also initiate the offloading by publishing the commit and split transaction of their respective ledger channels. This forces I to publish the commit and split transactions of the respective other ledger channel, since I loses her collateral to P otherwise.

More precisely, if I wishes to offload the virtual channel γ and retrieve her collateral and fees, she can close both of her ledger channels with A and B (i.e., α and β) and publish the funding transaction of the virtual channel i.e., $\mathtt{TX_f}$. This is possible as I is part of both ledger channels. A or B, on the other hand, are respectively part of only one ledger channel and hence they cannot offload the virtual channel individually. However, they can force I to offload by publishing the commit and split transactions of their respective channel with I (we will elaborate on this in the description of the punishment mechanism). Figure 7 illustrates the transactions that are posted on the blockchain in case of a successful offload. The figure shows that the split transactions of both underlying ledger channels have to be published such that eventually the funding transaction of the virtual channel can be published which completes the offloading procedure.

Punish Party $P \in \{A, B\}$ can punish I by taking all the coins on their respective ledger channels if the funding transaction of the virtual channel γ is not published on the ledger. In other words, it is I's responsibility to ensure that the state of her ledger channels with A and B are not updated while γ is open. Furthermore, upon one of the subchannels being closed, I must close the other subchannel in order to guarantee that both parties can post $\mathtt{TX_f}$.

Let us now get into more details. Assume that A's ledger channel with I is closed, but the funding transaction $\mathtt{TX_f}$ cannot be published on the blockchain. This means that I's channel with B (i.e., β) is still open or has been closed in a different state such that $\mathtt{TX_f}$ cannot be published. In other words, Ingrid acted maliciously by wrongfully closing β in a different state or by not closing β at all. In this case, A must be able to get all the coins from her channel with Ingrid. This punishment works as follows: After A publishing the split transaction of α, I is given a certain time period to close her channel with B and publish the virtual channel's funding transaction $\mathtt{TX_f}$. If I fails to do so in the prescribed time period, A receives all coins in her channel with I.

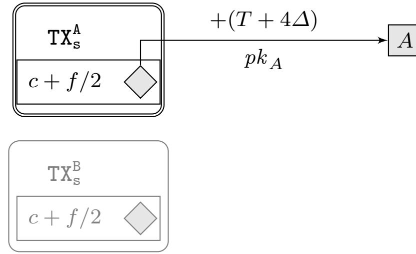

Fig. 8: Transactions published after A successfully executed the punishment procedure. The grayed transaction $\texttt{TX}_\texttt{s}^\texttt{B}$ indicates that this transaction has not been published.

We note that in this scenario, B (instead of I) might have been the malicious party by closing β in an outdated state, thereby leaving I no option to publish $\texttt{TX}_\texttt{f}$. However, in this case, I can punish B via the punishment mechanism of the underlying ledger channel and earn all the coins in β. Therefore, I will remain financially neutral as she gets punished by A but simultaneously compensated by B. Figure 8 illustrates the transactions that are posted on the blockchain in the case of A successfully executing the punishment mechanism. The case where B executes the punishment mechanism is analogous.

Further discussion regarding our constructions In the following, we present further considerations regarding our protocol, including remarks on concurrency, a discussion on how the protocol can be built on top of Lightning channels, and a brief description of our virtual channel construction with validity that we detail in Appendix B.

Concurrency When creating a virtual channel, we need to lock the underlying ledger channels α and β (i.e., no further updates can be made on the ledger channels as long as the virtual channel is open). This, however, is undesirable, because in most cases the ledger channels will have more coins available than what is needed for funding the virtual channel. We emphasize that this issue can be easily addressed (and hence supporting full concurrency) by using the channel splitting technique discussed in [3]. This means that before constructing the virtual channel Alice-Bob, parties would first *split* each underlying ledger channel off-chain in two channels: (i) one would contain the exact amount of coins for the virtual channel and (ii) the other one would contain the remaining coins that can be used in the underlying ledger channel.

Virtual channels over Lightning We will now discuss how our virtual channel constructions can be built on top of any ledger channel infrastructure that uses a *revocation/punishment* mechanism such as the Lightning Network [24]. The main complication arises from the fact that ledger channel constructions other than generalized channels require two commit transactions per channel state (one for each party). As depicted in Figure 9 (and unlike generalized channels in Figure 2), Alice and Bob

each have a commit transaction $\mathtt{TX}_\mathtt{c}^\mathtt{A}$ and $\mathtt{TX}_\mathtt{c}^\mathtt{B}$ which spends the funding transaction $\mathtt{TX}_\mathtt{f}$ and distributes the coins. Therefore, in such channel constructions, it is a priori unclear which of these commit transactions will be posted and accepted on the blockchain (note that only one of them can be successfully published) and hence building applications (e.g., virtual channels) on top of such ledger channels becomes complex.

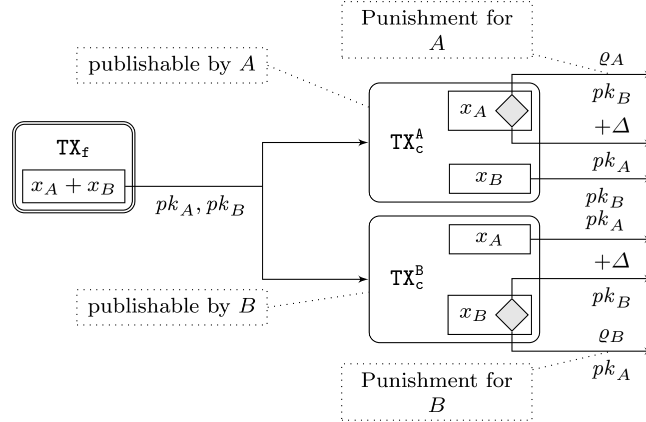

Fig. 9: A Lightning style payment channel where A has x_A coins and B has x_B coins. Δ upper bounds the time needed to publish a transaction on a blockchain. condition ϱ_A represents the verification of A' revocation secret and h represents the verification of B' revocation secret.

In more detail, assume Alice and Bob want to build a virtual channel γ on top of their respective Lightning ledger channels with Ingrid, where both ledger channels consist of two commit transactions respectively (i.e., $(\mathtt{TX}_\mathtt{c}^\mathtt{A}, \mathtt{TX}_\mathtt{c}^\mathtt{IA})$ for the channel between Alice and Ingrid and $(\mathtt{TX}_\mathtt{c}^\mathtt{B}, \mathtt{TX}_\mathtt{c}^\mathtt{IB})$ for the channel between Bob and Ingrid). All three parties now have to make sure that the virtual channel can be funded (i.e., that the funding transaction of γ can be published to the blockchain) even in case of malicious behavior. To ensure this, parties have to prepare the funding transaction of γ with respect to all possible combinations of the commit transactions of the respective underlying ledger channels. Since there are four such combinations $((\mathtt{TX}_\mathtt{c}^\mathtt{A}, \mathtt{TX}_\mathtt{c}^\mathtt{B})$, $(\mathtt{TX}_\mathtt{c}^\mathtt{A}, \mathtt{TX}_\mathtt{c}^\mathtt{IB})$, $(\mathtt{TX}_\mathtt{c}^\mathtt{IA}, \mathtt{TX}_\mathtt{c}^\mathtt{B})$ and $(\mathtt{TX}_\mathtt{c}^\mathtt{IA}, \mathtt{TX}_\mathtt{c}^\mathtt{IB}))$, parties have to prepare four funding transactions for γ. Hence, updating such a virtual channel requires repeating the update procedure for all four funding transactions.

As generalized channels require only a single commit transaction per channel state building virtual channels on top of generalized channels offers a significant efficiency improvement in terms of off-chain communication complexity (see Section 5 for the detailed comparison).

Virtual Channels With Validity Note that so far we described our protocol without validity where the virtual channel can be offloaded by the intermediary whenever she wants. The drawback of this construction is that Ingrid needs to be proactive during

the lifetime of the virtual channel, i.e., she has to constantly monitor the channel for potential misbehavior of Alice or Bob. This might be undesirable in scenarios where Ingrid plays the role of the intermediary in not just one but many different virtual channels at the same time (e.g., if Ingrid is a channel hub). For this reason, we developed an alternative solution which we call virtual channels with validity. In this solution, each virtual channel has a predetermined time (which we call validity) which indicates until when the channel has to be closed again. If the channel is still open after this time, Ingrid has to become proactive in order to receive her collateral back. The obvious advantage of this approach is that Ingrid can remain inactive until the validity of a channel expires. The details of this protocol can be found in Appendix B.

4 Security Model and Analysis

In order to model and prove the security of our virtual channel protocols, we use the global UC framework (GUC) [9] as in [3]. This framework allows for a global setup which we utilize to model a public blockchain. More precisely, our protocol uses a global ledger functionality $\widehat{\mathcal{L}}(\varDelta, \varSigma)$, where $\varDelta$ upper bounds the blockchain delay, i.e., the maximum number of rounds required to publish a transaction, and $\varSigma$ is the signature scheme used by the blockchain. In this section, we only give a high-level idea behind our security analysis in the UC framework and refer the readers to the full version of the paper [2] for more details.

As a first step, we define the expected behavior of a virtual channel protocol in the form of an *ideal functionality* $\mathcal{F}_V$. The functionality defines the input/output behavior of a protocol, its impact on the global setup (e.g., ledger) and the possible ways an adversary can influence its execution (e.g., delaying messages). In order to prove that a concrete protocol is a secure virtual channel protocol, one must show that the protocol *emulates* the ideal functionality $\mathcal{F}_V$. This means that any attack that can be mounted on the protocol can also be mounted on the ideal functionality, hence the protocol is at least as secure as the ideal specification given by $\mathcal{F}_V$.

The proof of emulation consists of two steps. First, one must design a *simulator*, which simulates the actions of an adversary on the real-world protocol by interacting with the ideal functionality. Second, it must be shown that the execution of the real-world protocol being attacked by a real-world adversary is indistinguishable from the execution of the ideal functionality communicating with the constructed simulator. In UC, the ppt distinguisher who tries to distinguish these two executions is called the *environment*.

The main challenge when designing a simulator is to make sure that the environment sees transactions being posted on the ledger in the same round in both worlds. In addition, our simulator needs to ensure that the ideal functionality outputs the same set of messages in the same round as the protocol. We reduce the indistinguishability of the two executions to the security of the cryptographic primitives used in our protocol.

One of the advantages of using UC is its composability. In other words, one can use an ideal functionality in a black-box way in other protocols. This simplifies the process of designing new protocols as it allows to reuse existing results and enables modular protocol designs. We utilize this nice property of the UC framework and use the ideal functionality of the generalized channel from [3] when designing our virtual channel protocol.

Due to lack of space, we only mention the main security theorem and provide a high-level proof sketch here. We refer the reader to the full version of this paper [2] for the full proof.

Theorem 1. *Let Σ be a signature scheme that is strongly unforgeable against chosen message attacks. Then for any ledger delay $\Delta \in \mathbb{N}$, the virtual channel protocol without validity as described in Section 3.4 working in $\mathcal{F}_{preL}(3,1)$-hybrid, UC-realizes the ideal functionality $\mathcal{F}_V(2)$.*

We now give a proof sketch to show that the two properties (V1) Balance security and (V2) Offload with punish hold for honest parties. To this end, we analyze all possible cases in which the underlying ledger channels are maliciously closed, i.e., the cases when the virtual channel cannot be offloaded anymore. Note that if the virtual channel is offloaded, it is effectively transformed into a generalized ledger channel and satisfies the security properties of generalized channels.

If all parties behave honestly (V1) and (V2) hold trivially as I is always able to offload the virtual channel by publishing all transactions $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$, $\mathtt{TX}_{\mathtt{s}}^{\mathtt{B}}$ and $\mathtt{TX}_{\mathtt{f}}$. Furthermore, neither A nor B would ever lose their coins. Now consider the case where one of the underlying channels, e.g., the channel between B and I is closed in a different state such that $\mathtt{TX}_{\mathtt{f}}$ cannot be posted on the blockchain anymore (the case for the channel between A and I is analogous). As an honest A would not update her channel with I as long as the virtual channel is open, there are only two possible situations: (i) A is able to post $\mathtt{TX}_{\mathtt{s}}^{\mathtt{A}}$ which allows her to punish I (see Figure 8), or (ii) I has maliciously closed her channel with A in an outdated and revoked state. In this case, A is able to punish I according to property (S3), i.e., instant finality with punish, of the underlying ledger channel (see Section 2 and Figure 2 for more details on the punishment of the underlying channel). Therefore, (V2) is satisfied for A, since she can punish I and get financially compensated. Now let us analyze the maliciously closed channel between B and I, let us denote it β. If both parties are malicious, we do not need to prove anything as (V1) and (V2) should only hold for honest parties. In case B is honest, I must have closed β in an old state which would allow B to punish I. Hence (V2) holds and we do not need to prove (V1) as I is malicious. Analogously, if I is honest, malicious B must have closed β in an old state and hence I can punish B. Hence (V1) holds and we do not need to prove (V2) for malicious B). Hence, (V1) and (V2) hold for all honest parties.

5 Performance evaluation

In this section, we first study the storage overhead on the blockchain as well as the communication overhead between users to use virtual channels. For each of these

aspects, we evaluate both constructions (i.e., with and without validity) built on top of both generalized channels as well as Lightning channels and compare them. Finally, we evaluate the advantages of virtual channels over ledger channels in terms of routing communication overhead and fee costs. As testbed [6], the transactions are created in Python using the library `python-bitcoin-utils` and the Bitcoin *Script* language. To showcase compatibility and feasibility, we deployed these transactions successfully on the Bitcoin testnet.

5.1 Communication overhead

We analyze the communication overhead imposed by the different operations, such as `CREATE`, `UPDATE`, `OFFLOAD` and `CLOSE`, by measuring the byte size of the transactions that need to be exchanged as well as the cost in USD necessary for posting the transactions that need to be published on-chain. The cost in USD is calculated by taking the price of 18803 USD per Bitcoin, and the average transaction fee of 104 satoshis per byte all of them at the time of writing. We detail in Table 2 the aforementioned costs measured for both virtual channel constructions building on top of generalized channels and on top of Lightning channels.

Operations	Generalized Channels										Lightning Channels									
	VC-NV					VC-V					VC-NV					VC-V				
	on-chain			off-chain		on-chain			off-chain		on-chain			off-chain		on-chain			off-chain	
	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size	# txs	size	cost	# txs	size
`CREATE`	0	0	0	7	2829	0	0	0	8	2803	0	0	0	16	7704	0	0	0	14	5722
`UPDATE`	0	0	0	2	695	0	0	0	2	695	0	0	0	8	2824	0	0	0	4	1412
`OFFLOAD`	5	2134	41.73	0	0	6	2108	41.22	0	0	3	1800	35.20	0	0	4	1778	34.77	0	0
`CLOSE` (opt)	0	0	0	4	1390	0	0	0	4	1390	0	0	0	4	1412	0	0	0	4	1412
`CLOSE` (pess)	7	2829	55.32	0	0	8	2803	54.81	0	0	4	2153	42.10	0	0	5	2131	41.67	0	0

Table 2: Evaluation of the virtual channels. For each operation we show: the number of on-chain and off-chain transactions (*# txs*) and their *size* in bytes. For on-chain transactions, *cost* is in USD and estimates cost of publish them on the ledger.

Perhaps the most relevant difference to ledger channels in practice is, in the `CREATE` and the optimistic `CLOSE` case, we do not have any on-chain transactions. This implies no on-chain fees for the opening and closing of virtual channels.

Virtual channels over generalized channels For the creation of a virtual channel (`CREATE` operation) on top of generalized channels, we need to update both ledger channels to a new state that can fund the virtual channel, requiring to exchange $2 \cdot 2$ transactions with 1494 (VC-NV) or 1422 (VC-V) bytes. Additionally, we need 640 bytes for $\mathtt{TX_f}$ (VC-NV) or 309 + 377 bytes for $\mathtt{TX_f}$ and $\mathtt{TX_{refund}}$ (VC-V). Finally, for both VC-NV and VC-V, we need the transactions representing the state of the the virtual channel itself which requires 431 bytes for $\mathtt{TX_c}$ and 264 bytes for $\mathtt{TX_s}$. This complete process results in 7 (VC-NV) or 8 (VC-V) transactions with a total of 2829 (VC-NV) or 2803 (VC-V) bytes. Forcefully closing (`CLOSE`(pess) operation) and offloading (`OFFLOAD` operation) requires the same set of transactions as with `CREATE`, minus the commitment and the split transaction (695 bytes) of the virtual channel in

the latter case, both on-chain. Finally, we observe that the UPDATE and the optimistic CLOSE(opt) operation require 2 transactions (695 bytes) for both constructions, as they are designed as an update of a ledger channel.

Virtual channels over Lightning channels Building virtual channels on top of Lightning channels yields the following results. Instead of one commitment and one split transaction per ledger channel, we now need two commitment transactions per ledger channel, each of size 580 (VC-NV) or 546 (VC-V) bytes. Due to the fact that in both ledger channels, either commitment transaction can be published, we now need four $\mathtt{TX_f}$ of 640 bytes each (VC-NV) or two $\mathtt{TX_f}$ of 309 and four $\mathtt{TX_{refund}}$ of 377 bytes (VC-V). For every $\mathtt{TX_f}$, we need two commitment transactions of 353 bytes (in total, $8 \cdot 353$ in VC-NV or $4 \cdot 353$ in VC-V). For OFFLOAD, only one commitment transaction per ledger channel needs to be published, along with one $\mathtt{TX_f}$ (for VC-NV) and $\mathtt{TX_f}$ plus $\mathtt{TX_{refund}}$ (for VC-V). CLOSE(pess), needs to publish a commitment transaction in addition to OFFLOAD, resulting in 2153 (VC-NV) or 2131 (VC-V) bytes.

5.2 Comparison to payment channel networks

In this section we compare virtual channels to multi-hop payments in a payment channel network (PCN). In a PCN, users route their payments via intermediaries. During the routing of a transaction tx, each intermediary party locks tx.cash coins as a "promise to pay" in their channels, a payment commitment that can technically be implemented as a Hash-Time Lock Contract (HTLC), e.g. as in the Lightning Network [24]. We now evaluate the difference in communication overhead and fee costs compared to virtual channels, summarize them in Table 3 and illustrate them in Figure 10.

Routing communication overhead When performing a payment between Alice and Bob via an intermediary Ingrid in a multi-hop payment over generalized channels, the participants need to update both generalized channels with a "promise to pay", which require 2 transactions or 818 bytes per channel when implemented as HTLC. If they are successful, both generalized channels need to be updated again to "confirm the payment" (again, 2 transactions or 695 bytes per channel). This whole process results in 8 transactions or $2 \cdot 818 + 2 \cdot 695 = 3026$ off-chain bytes that need to be exchanged. Generically, if the parties want to perform n sequential payments, they need to exchange $8 \cdot n$ transaction with a total of $3026 \cdot n$ bytes.

Assume now that Alice and Bob were to perform the payment over a virtual channel without validity instead and that this virtual channel is not yet created. As shown in Table 2, they need to open the virtual channel for 2829 bytes, where they set the balance of the virtual channel already to the correct state after the payment, and then close it again for 1390 bytes, resulting in a total of 4219 off-chain bytes. However, if we again consider n sequential payments, the result would be $9 + 2 \cdot n$ transactions or $3524 + 695 \cdot n$ bytes, which supposes a reduction of $2331 \cdot n - 3524$ bytes with respect to relying on generalized channels only. This means that a virtual channel is already cheaper if only two (or more) sequential

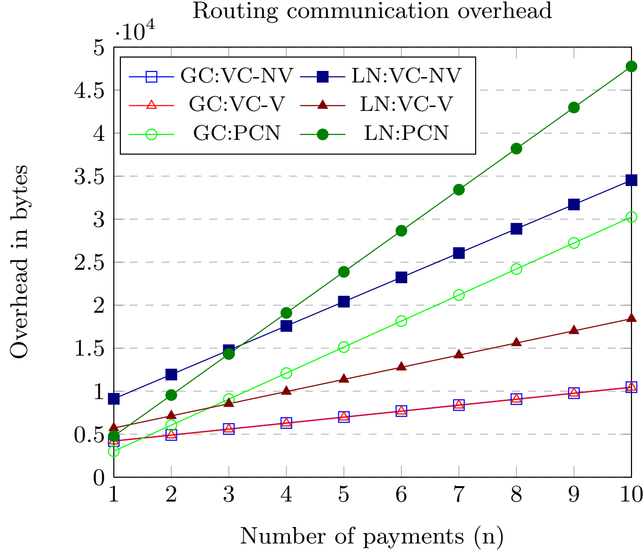

Fig. 10: Pictorial illustration of Table 3.

transactions are performed. We obtain similar results if we consider virtual channels with validity instead. For Lightning channels, the overhead is larger for both the multi-hop payment and the VC setting (Table 3).

	Overhead in bytes			fees
	1 paym.	2 paym.	n payments	tx.cash in n payments
GC: PCN	3026	6052	$3026 \cdot n$	$\mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$
GC: VC-NV	4219	4914	$3524 + 695 \cdot n$	$\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$
GC: VC-V	4193	4888	$3498 + 695 \cdot n$	
LN: PCN	4776	9552	$4776 \cdot n$	$\mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$
LN: VC-NV	9116	11940	$6292 + 2824 \cdot n$	$\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$
LN: VC-V	5722	7134	$4310 + 1412 \cdot n$	

Table 3: Comparison of virtual channels (VC) to multi-hop payments (PCN) showing the overhead in bytes for a different number of payments and the difference in fees.

Fee costs In a multi-hop payment tx in a PCN, the intermediary user Ingrid charges a base fee (BF) for being online and offering the routing service and relative fee (FR) for locking the amounts of coins ($\mathsf{tx.cash}$) and changing the balance in the channel, so that $\mathsf{fee}(\mathsf{tx}) := \mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$. Note that at the time of writing, the fees are $\mathsf{BF} = 1$ satoshi and $\mathsf{FR} = 0.000001$.

In a virtual channel setting, $\gamma.\mathsf{Ingrid}$ can charge a base fee to collaborate to open and close the virtual channel, and also a relative fee to lock collateral coins in the virtual channel. However, no fees per payment are charged by Ingrid as she does not participate in them (and even does not know how many end-users performed)1. Let us now investigate the case of paying $\mathsf{tx.cash}$ in n micropayments of equal value. In PCN case, the total cost would be $\sum_{i=1}^{n} \mathsf{BF} + \mathsf{FR} \cdot \frac{\mathsf{tx.cash}}{n} = \mathsf{BF} \cdot n + \mathsf{FR} \cdot \mathsf{tx.cash}$.

Whereas, in the virtual case, the parties first create a virtual channel γ with balance $\mathsf{tx.cash}$, and they will handle the micropayments in γ. Thereby, the cost would be only the opening cost of the virtual channel, for which we assumed $\mathsf{BF} + \mathsf{FR} \cdot \mathsf{tx.cash}$. Thus, if Alice and Bob would make more than one transaction, i.e., $n > 1$, it is beneficial to use virtual channels for reducing the fee costs by $\mathsf{BF} \cdot (n - 1)$.

Summary We find that the best construction in practice is the combination of virtual channels on top of generalized channels, as this yields the least overhead after only two or more sequential payments. However, building virtual channels over LN channels also yields less overhead than multi-hop PCN payments over LN.

6 Related Work

In this section, we position this work in the landscape of the literature for off-chain payments protocols.

Payment Channels Started from the Lightning Channels construction [24], the idea of 2-party payment channels has been largely used in academia and industry as a building block for more complex off-chain payment protocols. More recently, Aumayr et al. [3] have proposed a novel construction for 2-party payment channels that overcome some of the drawbacks of the original Lightning channels. While their benefit in terms of scalability is out of any doubt by now, payment channels are limited to payments between two users and consequently its overall utility.

A concurrent work [17] has also proposed a virtual channel construction over Bitcoin. However, their construction uses decreasing time-locks instead of a punishment mechanism in order to guarantee that only the latest state can be posted on the blockchain. As a consequence, their construction only allows a fixed number of transactions to be made during the lifetime of the virtual channel. This is quite restrictive as it requires users to close and open new virtual channels more frequently which goes against the purpose of virtual channels. Note that one cannot simply increase the time-lock as this would essentially lock the coins of the users for a longer period of time. Furthermore, our constructions are *generalized* virtual channels, i.e., they are not limited to just payments, but rather allow to run any Bitcoin script off-chain. In addition, we propose a modular approach compared to the monolithic construction in [17]. Finally, our work proposes two protocols, which each have their advantages in different use cases.

Payment Channel Networks (PCN) and Payment Channel Hub (PCH) A PCN allows a payment between two users that do not share a payment channel but are however connected through a path of payment channels. The notion of PCN started with the deployment of Lightning Network [24] for Bitcoin and Raiden Network [28] for Ethereum and has been widely studied in academia to research into different aspects such as privacy [19, 20], routing of payments [25], collateral management [14] and others. Similar to PCN, different constructions for PCH exist [26, 16, 5] that allow a payment between two users through a single intermediary, the payment hub.

PCNs and PCHs, however, share the drawback that each payment between two users require the active involvement of the intermediary (or several intermediaries in the case of PCH), which reduces the reliability (e.g., the intermediary can go offline) and increases the cost of the payment (e.g., each intermediary charges a fee for the payment).

State Channels Several works [11, 13, 21, 10] have shown how to leverage the highly expressive scripting language available at Ethereum to construct (multi-party) state channels. A state channel allows the involved parties to carry out off-chain computations, possibly other than payments. Closer to our work, Dziembowski et al. [12] showed how to construct a virtual channel leveraging two payment channels defined in Ethereum. These approaches are, however, highly tight to the functionality provided by the Ethereum scripting language and their constructions cannot be reused in other cryptocurrencies. In this work, we instead show that virtual channels can be constructed from digital signatures and timelock mechanism only, which makes virtual channels accessible for virtually any cryptocurrency system available today.

7 Conclusion

Current PCNs route payments between two users through intermediate nodes, making the system less reliable (intermediaries might be offline), expensive (intermediaries charge a fee per payment), and privacy-invasive (intermediate nodes observe every payment they route). To mitigate this, recent work has introduced the concept of virtual channels, which involve intermediaries only in the creation of a bridge between payer and payee, who can later on independently perform arbitrarily many off-chain transactions. Unfortunately, existing constructions are only available for Ethereum, as they rely on its account model and Turing-complete scripting language.

In this work, we present the first virtual channel constructions that are built on the UTXO-model and require a scripting language supported by virtually every cryptocurrency, including Bitcoin. Our two protocols provide a tradeoff on who can offload the virtual channel, similar to the preemptible vs. non-preemptible virtual machines in the cloud setting. In other words, our virtual channel construction without validity is more suitable for intermediaries who can monitor the blockchain regularly, such as payment channel hubs, but can also close the virtual channel at anytime if desired. Our virtual channel protocol with validity however, is more suitable for light intermediaries who do not wish to be active during the lifetime of the virtual channel but cannot close the virtual channel before its validity has expired. We formalize the security properties of virtual channels in the UC framework, proving that our protocols constitute a secure realization thereof. We have prototyped our protocols and evaluated their efficiency: for n sequential payments in the optimistic case, they require $9 + 2 \cdot n$ off-chain transactions for a total of $3524 + 695 \cdot n$ bytes, with no on-chain footprint.

As mentioned in the introduction of this work, the task of designing secure virtual channels has been proven to be challenging even on a cryptocurrency like

Ethereum [12] which supports smart contract execution. Unsurprisingly, this task becomes even more complex when building virtual channels for blockchains that support only a limited scripting language as it is not possible to take advantage of the full computation power of Turing complete smart contracts. Due to these significantly differing underlying assumptions (smart contracts vs. limited scripting languages), the virtual channel protocols based on Ethereum [12] and the protocols presented in this work are incomparable. We emphasize that we view our virtual channel constructions as complementary to the one presented in [12], as we do not aim to improve the construction of [12] but rather extend the concept of virtual channels to a broader class of blockchains.

We conjecture that it is possible to recursively build virtual channels on top of any two underlying channels (either ledger, virtual or a combination of them), requiring to adjust the timings for offloading channels: users of a virtual channel at layer k should have enough time to offload the (virtual/ledger) channels at layers 1 to $k-1$. Additionally, we envision that while virtual channels without validity might serve as a building block at any layer of recursion, virtual channels with validity period may be more suitable for the top layer as they have a predefined expiration time after which they would require to offload in any case all underlying layers. We plan to explore the recursive building of virtual channels in the near future. Additionally, we conjecture that virtual channels help with privacy, but we leave a formalization of this claim as interesting future work, as it involves a quantitative analysis that falls off the scope of this work.

Acknowledgments

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

References

[1] A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Crypto-Currencies.* 1st. O'Reilly Media, Inc., 2014. ISBN: 1449374042.

[2] L. Aumayr et al. *Bitcoin-Compatible Virtual Channels*. Cryptology ePrint Archive, Report 2020/554. https://eprint.iacr.org/2020/554. 2020.

[3] L. Aumayr et al. *Generalized Bitcoin-Compatible Channels*. Cryptology ePrint Archive, Report 2020/476. https://eprint.iacr.org/2020/476. 2020.

[4] S. Bano et al. "SoK: Consensus in the Age of Blockchains". In: *AFT 2019*, pp. 183–198.

[5] E. Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *IEEE SP*. 2014, pp. 459–474.

[6] *Bitcoin-Compatible Virtual Channels: Github repository*. https://github.com/utxo-virtual-channels/vc. 2020.

[7] *Bitcoin Wiki: Payment Channels*. https://en.bitcoin.it/wiki/Payment_channels. 2018.

[8] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145.

[9] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. "Universally Composable Security with Global Setup". In: *TCC 2007*. Ed. by S. P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, Feb. 2007, pp. 61–85.

[10] M. M. T. Chakravarty et al. "Hydra: Fast Isomorphic State Channels". In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 299. URL: https://eprint.iacr.org/2020/299.

[11] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. "Multi-party Virtual State Channels". In: *EUROCRYPT 2019, Part I*. 2019, pp. 625–656.

[12] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. "Perun: Virtual Payment Hubs over Cryptocurrencies". In: *IEEE SP*. 2019, pp. 106–123.

[13] S. Dziembowski, S. Faust, and K. Hostakova. "General State Channel Networks". In: *ACM CCS*. 2018, pp. 949–966.

[14] C. Egger, P. Moreno-Sanchez, and M. Maffei. "Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks". In: *ACM CCS*. 2019, pp. 801–815.

[15] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. "SoK: Layer-Two Blockchain Protocols". In: *FC 2020*. 2020, pp. 201–226.

[16] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. "TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub". In: *NDSS 2017*. The Internet Society, 2017.

[17] M. Jourenko, M. Larangeira, and K. Tanaka. "Lightweight Virtual Payment Channels". In: *CANS 2020*. 2020, pp. 365–384.

[18] G. Kappos et al. "An Empirical Analysis of Privacy in the Lightning Network". In: *CoRR* abs/2003.12470 (2020). arXiv: 2003.12470. URL: https://arxiv.org/abs/2003.12470.

[19] G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi. "Concurrency and Privacy with Payment-Channel Networks". In: *ACM CCS 17*. Ed. by B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu. ACM Press, 2017, pp. 455–471.

[20] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei. "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability". In: *NDSS*. 2019.
[21] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. "Sprites and State Channels: Payment Networks that Go Faster Than Lightning". In: *FC 2019*. 2019, pp. 508–526.
[22] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. `http://bitcoin.org/bitcoin.pdf`. 2009.
[23] U. Nisslmueller, K. Foerster, S. Schmid, and C. Decker. "Toward Active and Passive Confidentiality Attacks on Cryptocurrency Off-chain Networks". In: *ICISSP*. Ed. by S. Furnell, P. Mori, E. R. Weippl, and O. Camp. 2020, pp. 7–14.
[24] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. Draft version 0.5.9.2, available at `https://lightning.network/lightning-network-paper.pdf`. Jan. 2016.
[25] S. Roos, P. Moreno-Sanchez, A. Kate, and I. Goldberg. "Settling Payments Fast and Private: Efficient Decentralized Routing for Path-Based Transactions". In: *NDSS*. 2018.
[26] E. Tairi, P. Moreno-Sanchez, and M. Maffei. *A^2L: Anonymous Atomic Locks for Scalability and Interoperability in Payment Channel Hubs*. In press.
[27] S. Tikhomirov, P. Moreno-Sanchez, and M. Maffei. "A Quantitative Analysis of Security, Anonymity and Scalability for the Lightning Network". In: *IEEE EuroS&P*. 2020, pp. 387–396.
[28] *Update from the Raiden team on development progress, announcement of raidEX*. `https://tinyurl.com/z2snp9e`. Feb. 2017.
[29] A. Zamyatin et al. *SoK: Communication Across Distributed Ledgers*. In press.

A Ideal functionality for virtual channels

Here we define the ideal functionality $\mathcal{F}_V$ that describes the ideal behavior of both ledger and virtual channels. The full description of the ideal functionalities can be found in the full version of this paper [2].

$\mathcal{F}_V$ can be viewed as an extension of the ledger channel functionality $\mathcal{F}_L$ defined in [3]. The functionality $\mathcal{F}_V$ is parameterized by a parameter T which upper bounds the maximum number of off-chain communication rounds between two parties required for any of the operations in $\mathcal{F}_L$. The ideal functionality $\mathcal{F}_V$ communicates with the parties $\mathcal{P}$, the simulator $\mathcal{S}$ and the ledger $\widehat{\mathcal{L}}$. It maintains a channel space Γ where it stores all currently opened ledger channels (together with their funding transaction tx) and virtual channels. Before we define $\mathcal{F}_V$ formally, we describe it at a high level.

Messages related to ledger channels For any message related to a ledger channel, $\mathcal{F}_V$ behaves as the functionality $\mathcal{F}_L$. That is, the corresponding code of $\mathcal{F}_L$ is executed when a message about a ledger channel γ is received. For the rest of this section, we discuss the behavior of $\mathcal{F}_V$ upon receiving a message about a virtual channel.

Create The creation of a virtual channel is equivalent to synchronously updating two ledger channels. Therefore, if all parties, namely γ.Alice, γ.Bob and γ.Ingrid, follow the protocol, i.e., update their ledger channels correctly, a virtual channel is successfully created. This is captured in the "All agreed" case of the functionality. Hence, if all parties send the `CREATE` message, the functionality returns `CREATED` to γ.users, keeps the underlying ledger channels locked and adds the virtual channel to its channel space Γ.

On the other hand, the creation of the virtual channel fails if after some time at least one of the parties does not send `CREATE` to the functionality. There are three possible situations: (i), the update is peacefully rejected and parties simply abort the virtual channel creation, (ii) both channels are forcefully closed, in order to prevent a situation where one of the channels is updated and the other one is not, (iii) if γ.Ingrid has not published the old state of one of her channels to the ledger after Δ rounds, it forcefully closes the ledger channels using the new state i.e., where γ.Ingrid behaves maliciously and can publish both the old and new states, while γ.Alice or γ.Bob can only publish the new state.

Update The update procedure for the virtual channel works in the same way as for ledger channels except in case of any disputes during the execution, the functionality calls `V–ForceClose` instead of `L–ForceClose`.

Offload We consider two types of offloading depending on whether the virtual channel is with or without validity. In the first case, offloading is initiated by one of the γ.endUsers before round γ.val, while for channels without validity, Ingrid can initiate the offloading at any time. Since offloading a virtual channel requires closure of the underlying subchannels, the functionality merely checks if either funding transaction of γ.subchan has been spent until round $T_1 + \Delta$. If not, the functionality outputs a message (`ERROR`). As in to [3], the `ERROR` message represents an impossible situation which should not happen as long as one of the parties is honest.

Close - channels without validity Upon receiving (`CLOSE`, id) from all parties in γ.users within $T_1 \leq 6T$ rounds (where the exact value of T_1 is specified by $\mathcal{S}$), all parties have peacefully agreed on closing the virtual channel, which is indicated by the "All Agreed" case. In this case the final balance of the parties is reflected on their underlying channels. When the update of Γ is completed, the ideal functionality sends `CLOSED` to all users. Due to the peaceful closure in this "All Agreed" case, the functionality defines property (E3).

If one of the (`CLOSE`, id) messages was not received within T_1 rounds ("Wait for others" case), the closing procedure fails. The following cases my happen: (i) the update procedure of an underlying ledger channel was aborted prematurely by γ.Alice or γ.Bob which would cause the virtual channel to be forcefully closed. (ii) γ.Ingrid refuses to revoke her state during the update of either one of the underlying ledger channels where the functionality waits Δ rounds and if γ.Ingrid has not published the old state to the ledger the functionality forcefully closes the ledger channels using the new state.

Close - channels with validity. This procedure starts in round $\gamma.\mathsf{val} - (4\Delta + 7T)$ to have enough time to forcefully close the channel if necessary. If within $T_1 \leq 6T$ rounds (where the exact value of T_1 is specified by $\mathcal{S}$) all $\gamma.\mathsf{users}$ agreed on closing the channel or if the simulator instructs the functionality to close the channel, the same steps as in the all agreed case for channels without validity are executed. Otherwise, after T_1 rounds, the functionality executes the forceful closure of the virtual channel.

Punish The punishment procedure is executed at the end of each round. It checks for every virtual channel γ if any of $\gamma.\mathsf{subchan}$ has just been closed and distinguishes if the consequence of closure was offloading or punishment. If after T_1 rounds (T_1 is set by $\mathcal{S}$) two transactions tx_1 and tx_2 are published on the ledger, where tx_1 refunds the collateral $\gamma.\mathsf{cash} + \gamma.\mathsf{fee}$ to $\gamma.\mathsf{Ingrid}$ and tx_2 funds γ on-chain, then the virtual channel has been offloaded and the message (`OFFLOADED`) is sent to $\gamma.\mathsf{users}$. If after T_1 rounds, only one transaction tx is on the ledger, which assigns $\gamma.\mathsf{cash}$ coins to a single honest party P and spends the funding transaction of only one of $\gamma.\mathsf{subchan}$, the functionality sends (`PUNISHED`) to P. Otherwise, the functionality outputs (`ERROR`) to $\gamma.\mathsf{users}$.

Notation In the functionality description, we use the notion of *rooted transactions* that we now explain (see Figure 11 for a concrete example). UTXO based blockchains can be viewed as a directed acyclic graph, where each node represents a transaction. Nodes corresponding to transactions tx_i and tx_j are connected with an edge if at least one of the outputs of tx_i is an input of tx_j, i.e, tx_i is (partially) funding tx_j. We denote the transitive reachability relation between nodes, which constitutes a partial order, as $\leq$. We say that a transaction tx is *rooted* in the set of transactions R if

1. $\forall \mathsf{tx}_i \leq \mathsf{tx}. \exists \mathsf{tx}_j \in R. \mathsf{tx}_j \leq \mathsf{tx}_i \vee \mathsf{tx}_i \leq \mathsf{tx}_j$,
2. $\forall \mathsf{tx}_i, \mathsf{tx}_j \in R. \mathsf{tx}_i \neq \mathsf{tx}_j, \mathsf{tx}_i \not\leq \mathsf{tx}_j$ and
3. $\mathsf{tx} \notin R$.

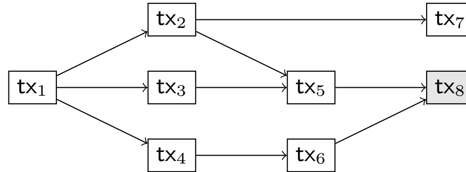

Fig. 11: The root sets of transaction tx_8 are $\{\mathsf{tx}_1\}$, $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_4\}$, $\{\mathsf{tx}_5, \mathsf{tx}_6\}$, $\{\mathsf{tx}_4, \mathsf{tx}_5\}$ and $\{\mathsf{tx}_2, \mathsf{tx}_3, \mathsf{tx}_6\}$.

Moreover, in order to simplify the notation in the functionality description, we write $m \overset{t}{\hookrightarrow} P$ as a short hand form for "send the message m to party P in round t." and $m \overset{t}{\hookleftarrow} P$ for "receive a message m from party P in round t".

Ideal Functionality $\mathcal{F}_V(T)$

Below we abbreviate $A := \gamma.\mathsf{Alice}$, $B := \gamma.\mathsf{Bob}$, $I = \gamma.\mathsf{Ingrid}$. For $P \in \gamma.\mathsf{endUsers}$, we denote $Q := \gamma.\mathsf{otherParty}(P)$.

For messages about ledger channels, behave as $\mathcal{F}_L(T, 1)$.

Create

Upon $(\texttt{CREATE}, \gamma) \overset{\tau}{\hookleftarrow} P$, let $\mathcal{S}$ define $T_1 \leq 8T$. If $P \in \gamma.\mathsf{endUsers}$, then define a set S, where $S := \{id_P\} := \gamma.\mathsf{subchan}(P)$, otherwise define S as $S := \{id_P, id_Q\} := \gamma.\mathsf{subchan}$. Lock all channels in S and distinguish:

All agreed: If you already received both $(\texttt{CREATE}, \gamma) \overset{\tau_1}{\hookleftarrow} Q_1$ and $(\texttt{CREATE}, \gamma) \overset{\tau_2}{\hookleftarrow} Q_2$, where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ and $\tau - T_1 \leq \tau_1 \leq \tau_2$, then in round $\tau_3 := \tau_1 + T_1$ proceed as:
1. Let $\mathcal{S}$ define $\vec{\theta_A}$ and $\vec{\theta_B}$ and set $(id_A, id_B) := \gamma.\mathsf{subchan}$.
2. Execute $\texttt{UpdateState}(id_A, \vec{\theta_A})$, $\texttt{UpdateState}(id_B, \vec{\theta_B})$, set $\Gamma(\gamma.\mathsf{id}) := \gamma$, send $(\texttt{CREATED}, \gamma) \overset{\tau_3}{\hookrightarrow} \gamma.\mathsf{endUsers}$, stop.

Wait for others: Else wait for at most T_1 rounds to receive $(\texttt{CREATE}, \gamma) \overset{\tau_1 \leq \tau + T_1}{\longleftarrow} Q_1$ and $(\texttt{CREATE}, \gamma) \overset{\tau_2 \leq \tau + T_1}{\longleftarrow} Q_2$ where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ (in that case option "All agreed" is executed). If at least one of those messages does not arrive before round $\tau + T_1$, do the following. For all $id_i \in S$, let $(\gamma_i, \mathsf{tx}_i) := \Gamma(id_i)$ and distinguish the following cases:
- If $\mathcal{S}$ sends (peaceful–reject, id_i), unlock id_i and stop.
- If $\gamma.\mathsf{Ingrid}$ is honest or if instructed by $\mathcal{S}$, execute $\texttt{L–ForceClose}(id_i)$ and stop.
- Otherwise wait for Δ rounds. If tx_i still unspent, then set $\vec{\theta}_{old} := \gamma_i.\mathsf{st}$, $\gamma_i.\mathsf{st} := \{\vec{\theta}_{old}, \vec{\theta}\}$ and $\Gamma(id_i) := (\gamma_i, \mathsf{tx}_i)$. Execute $\texttt{L–ForceClose}(id_i)$ and stop.

Update

Upon $(\texttt{UPDATE}, id, \vec{\theta}, t_{\mathsf{stp}}) \overset{\tau_0}{\hookleftarrow} P$, where $P \in \gamma.\mathsf{endUsers}$, behave as $\mathcal{F}_L(T, 1)$ yet replace the calls to $\texttt{L–ForceClose}$ in $\mathcal{F}_L(T, 1)$ with calls to $\texttt{V–ForceClose}$.

Offload

Upon $(\texttt{OFFLOAD}, id) \overset{\tau_0}{\hookleftarrow} P$, execute $\texttt{Offload}(id)$.

Close

Channels without validity:

Upon $(\texttt{CLOSE}, id) \overset{\tau}{\hookleftarrow} P$, where $\gamma(id).\mathsf{val} = \bot$, let $\mathcal{S}$ define $T_1 \leq 6T$. If $P \in \gamma_i.\mathsf{endUsers}$, define a set S, where $S := \{id_P\} := \gamma_i.\mathsf{subchan}(P)$, else define S as $S := \{id_P, id_Q\} := \gamma_i.\mathsf{subchan}$ and distinguish:

All agreed: If you received both messages $(\texttt{CLOSE}, id) \overset{\tau_1}{\hookleftarrow} Q_1$ and $(\texttt{CLOSE}, id) \overset{\tau_2}{\hookleftarrow} Q_2$, where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ and $\tau - T_1 \leq \tau_1 \leq \tau_2$, then in round $\tau_3 := \tau_1 + T_1$ proceed as follows:
1. Let $\gamma := \Gamma(id)$, $(id_A, id_B) := \gamma.\mathsf{subchan}$.
2. Parse $\gamma.\mathsf{st} = \{(c_A, \texttt{One–Sig}_A), (c_B, \texttt{One–Sig}_B)\}$ and set
$$\vec{\theta_A} := ((c_A, \texttt{One–Sig}_A), (c_B + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_I)),$$
$$\vec{\theta_B} := ((c_A + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_I), (c_B, \texttt{One–Sig}_B)),$$
3. Unlock both subchannels and execute $\texttt{UpdateState}(id_A, \vec{\theta_A})$ and $\texttt{UpdateState}(id_B, \vec{\theta_B})$. And set $\Gamma(id) := \bot$ and send $(\texttt{CLOSED}, \gamma) \overset{\tau_3}{\hookrightarrow} \gamma.\mathsf{endUsers}$.

Wait for others: Else wait for at most T_1 rounds to receive $(\texttt{CLOSE}, \gamma) \xleftarrow{\tau_1 \leq \tau + T_1} Q_1$ and $(\texttt{CLOSE}, \gamma) \xleftarrow{\tau_2 \leq \tau + T_1} Q_2$ where $Q_1, Q_2 \in \gamma.\mathsf{users} \setminus \{P\}$ (in that case option "All agreed" is executed). For all $id_i \in S$ let $(\gamma_i, \mathsf{tx}_i) := \Gamma(id_i)$, if such messages are not received until round $\tau + T_1$, set $\vec{\theta}_{old} := \gamma'.\mathsf{st}$ and distinguish:

- If $\gamma.\mathsf{Ingrid}$ is honest or if instructed by $\mathcal{S}$, execute $\texttt{V–ForceClose}(id_i)$ and stop.
- Else wait for Δ rounds. If tx_i still unspent, set $\gamma_i.\mathsf{st} := \{\vec{\theta}_{old}, \vec{\theta}\}$ and $\Gamma(id_i) := (\gamma_i, \mathsf{tx}_i)$. Execute $\texttt{L–ForceClose}(id_i)$ and stop.

Channels with validity:
For every $\gamma \in \Gamma$ s.t. $\gamma.\mathsf{val} \neq \bot$, in round $\tau_0 := \gamma.\mathsf{val} - (4\Delta + 7T)$ proceed as follows: let $\mathcal{S}$ set $T_1 \leq 6T$ and distinguish:

Peaceful close: If all parties in $\gamma.\mathsf{users}$ are honest or if instructed by $\mathcal{S}$, execute steps (1)–(3) of the "All agreed" case for channels without validity with $\tau_3 := \tau_0 + T_1$.
Force close: Else in round τ_3 execute $\texttt{V–ForceClose}(\gamma.\mathsf{id})$.

Punishment (executed at the end of every round)

For every id, where $\gamma := \Gamma(id)$ is a virtual channel, set $(id_A, id_B) := \gamma.\mathsf{subchan}$. If this is the first round when $\Gamma(id_A) = (\bot, \mathsf{tx}_A)$ or $\Gamma(id_B) = (\bot, \mathsf{tx}_B)$, i.e., one of the subchannels was just closed, then let $\mathcal{S}$ set $t_1 \leq T'$, where $T' := \tau_0 + T + 5\Delta$ if $\gamma.\mathsf{val} = \bot$ and $T' := \gamma.\mathsf{val} + 3\Delta$ if $\gamma.\mathsf{val} \neq \bot$, and distinguish the following cases:

Offloaded: Latest in round t_1 the ledger $\widehat{\mathcal{L}}$ contains both

- a transaction tx_1 rooted at $\{\mathsf{tx}_A, \mathsf{tx}_B\}$ with an output $(\gamma.\mathsf{cash} + \gamma.\mathsf{fee}, \texttt{One–Sig}_I)$. In this case $(\texttt{OFFLOADED}, id) \xhookrightarrow{\tau_1} I$, where τ_1 is the round tx_1 appeared on $\widehat{\mathcal{L}}$.
- a transaction tx_2 with an output of value $\gamma.\mathsf{cash}$ and rooted at $\{\mathsf{tx}_A, \mathsf{tx}_B\}$, if $\gamma.\mathsf{val} = \bot$, and rooted at $\{\mathsf{tx}_A\}$, if $\gamma.\mathsf{val} \neq \bot$. Let τ_2 be the round when tx_2 appeared on $\widehat{\mathcal{L}}$. Then output $(\texttt{OFFLOADED}, id) \xhookrightarrow{\tau_2} \gamma.\mathsf{endUsers}$, set $\gamma' = \gamma$, $\gamma'.\mathsf{Ingrid} = \bot$, $\gamma'.\mathsf{subchan} = \bot$, $\gamma.\mathsf{val} = \bot$ and define $\Gamma(id) := (\gamma', \mathsf{tx}_2)$.

Punished: Else for every honest party $P \in \gamma.\mathsf{users}$, check the following: the ledger $\widehat{\mathcal{L}}$ contains in round $\tau_1 \leq t_1$ a transaction tx rooted at either tx_A or tx_B with $(\gamma.\mathsf{cash} + \gamma.\mathsf{fee}/2, \texttt{One–Sig}_P)$ as output. In that case, output $(\texttt{PUNISHED}, id) \xhookrightarrow{\tau_1} P$. Set $\Gamma(id) = \bot$ in the first round when $\texttt{PUNISHED}$ was sent to all honest parties.
Error: If the above case is not true, then $(\texttt{ERROR}) \xhookrightarrow{t_1} \gamma.\mathsf{users}$.

$\texttt{V–ForceClose}(id)$: Let τ_0 be the current round and $\gamma := \Gamma(id)$. Execute subprocedure $\texttt{Offload}(id)$. Let $T' := \tau_0 + 2T + 8\Delta$ if $\gamma.\mathsf{val} = \bot$ and $T' := \gamma.\mathsf{val} + 3\Delta$ if $\gamma.\mathsf{val} \neq \bot$. If in round $\tau_1 \leq T'$ it holds that $\Gamma(id) = (\gamma, \mathsf{tx})$, execute subprocedure $\texttt{L–ForceClose}(id)$.
Subprocedure $\texttt{Offload}(id)$: Let τ_0 be the current round, $\gamma := \Gamma(id)$, $(id_\alpha, id_\beta) := \gamma.\mathsf{subchan}$, $(\alpha, \mathsf{tx}_A) := \Gamma(id_\alpha)$ and $(\beta, \mathsf{tx}_B) := \Gamma(id_\beta)$. If within Δ rounds, neither tx_A nor tx_B is spent, then output $(\texttt{ERROR}) \xhookrightarrow{\tau_0 + \Delta} \gamma.\mathsf{users}$.
Subprocedure $\texttt{UpdateState}(id, \vec{\theta})$: Let $(\alpha, \mathsf{tx}) := \Gamma(id)$. Set $\alpha.\mathsf{st} := \vec{\theta}$ and update $\Gamma(id) := (\alpha, \mathsf{tx})$.

B Concrete Instantiation With Validity

We now briefly present our virtual channel protocol with validity. We focus mainly on the creation of the virtual channel as this illustrates the main structural differences to our construction without validity.

Create Unlike the without validity case, the structure of the construction with validity is not symmetric (see Figure 12). The output of the ledger channel between A and I is used as the input for the funding transaction of the virtual channel $\texttt{TX}_\texttt{f}$,

whereas the output of the channel between B and I is used for the so-called refund transaction $\mathtt{TX_{refund}}$.

A can create $\mathtt{TX_f}$ on her own from the last state of her ledger channel with I. As a second step, A and B can already create the transactions required for the virtual channel γ. Additionally, I and B create the refund transaction which returns I's collateral if the virtual channel is offloaded. Finally, the created transactions are signed in reverse order. In particular, B signs $\mathtt{TX_{refund}}$ so that I is ensured that she can publish it and receive her collateral and fees. Then, I signs $\mathtt{TX_f}$ and provides the signature to A, effectively authorizing her to publish $\mathtt{TX_f}$, thereby allowing A to offload the virtual channel.

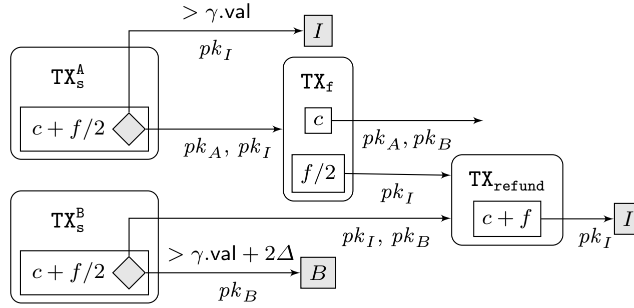

Fig. 12: Funding of a virtual channel γ with validity $\gamma.\mathsf{val}$.

Offload In our virtual channel with validity, only A can offload the virtual channel γ by publishing the commit and split transaction of her ledger channel with I. Although I and B are not able to offload the virtual channel, they have the guarantee that after round $\gamma.\mathsf{val}$ either the channel is offloaded or closed or they can punish A and get reimbursed.

Punish Recall that after a successful offload, the punishment mechanisms of generalized channels apply. We now discuss other malicious behaviors specific to this construction. In this protocol, only A can post the funding transaction of the virtual channel. If the virtual channel is not closed or offloaded by $\gamma.\mathsf{val}$, A is punished. A loses her coins to I and I loses her coins to B. Therefore, though B cannot offload the channel, he will get reimbursed from his ledger channel with I and I will get reimbursed regardless of whether the virtual channel is offloaded or not. At the time val, if the virtual channel is not honestly closed or the funding is not published, I submits the punishment transaction to reimburse her collateral. Therefore, at time $\mathsf{val} + \Delta$, either the punishment or the funding transaction is posted. If the virtual channel is offloaded, I can publish the refund transaction within Δ to get her coins back.

C Protocol Pseudocode

In Figure 13, we present the pseudocode of our modular virtual channel protocol that was described at a high level in Section 3.4.

Create virtual channel for $P \in \{A, B\}$

// Initiate creation of γ with funding source tid_A, tid_B in round t_0
Let γ_P be the channel with id $\gamma.\mathsf{subchan}(P)$.
Compute $\theta_P := \mathtt{GenVChannelOutput}(\gamma_P, P)$
Assign $\mathsf{Setup} \leftarrow \mathtt{SetupVChannel}^P(\gamma, tid_A, tid_B)$
if $\mathsf{UpdateChan}^P(\gamma_P, \theta_P, \mathsf{Setup})$ returns $\mathtt{UPDATE\text{-}OK}$
Creation successful.

Close virtual channel for $P \in \{A, B\}$

// Initiate closure of γ in round t_0^P
Let γ_P be the channel with id $\gamma.\mathsf{subchan}(P)$.
Parse $\gamma.\mathsf{st} = \left((c_P, \mathtt{One\text{-}Sig}_{pk_P}), (c_Q, \mathtt{One\text{-}Sig}_{pk_Q})\right)$
Compute $\vec{\theta}_P := \{(c_P, \mathtt{One\text{-}Sig}_{pk_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \mathtt{One\text{-}Sig}_{pk_I})\}$
if $\mathsf{UpdateChan}^P(\gamma_P, \vec{\theta}_P, \bot)$ returns $\mathtt{UPDATE\text{-}OK}$
Close successful.
else Execute $\mathtt{Offload}^P(\gamma)$ and stop.

Update virtual channel for $P \in \{A, B\}$

// Initiate update of γ with state $\vec{\theta}$ in t_0^P
if $\mathsf{UpdateChan}^P(\gamma, \vec{\theta}, \bot)$ returns $\mathtt{UPDATE\text{-}OK}$
Update successful.
else Execute $\mathtt{Offload}^P(\gamma)$ and stop.

$\mathtt{SetupVChannel}(\gamma, tid_A, tid_B)$

// Return the setup procedure Setup required for the setup of the virtual channel.
// The funding transaction and initial versions of split and commit transactions of
// the virtual channel γ are created. Moreover, the punishment and refund
// transactions are generated to be used in malicious cases.

$\mathtt{GenVChannelOutput}(\gamma_P, P)$

// Return output θ of γ_P that will fund the virtual channel

Create virtual channel for I

// React to creation of γ with funding source tid_A, tid_B in round t_0^I
Let γ_P be the channel with id $\gamma.\mathsf{subchan}(P)$ for $P \in \{A, B\}$.
Compute $\theta_P := \mathtt{GenVChannelOutput}(\gamma_P, P)$ for $P \in \{A, B\}$
Assign $\mathsf{Setup} \leftarrow \mathtt{SetupVChannel}^I(\gamma, tid_A, tid_B)$
if $\mathsf{UpdateChanSync}^I(\gamma_A, \vec{\theta}_A, \gamma_B, \vec{\theta}_B, \mathsf{Setup})$ returns
$\mathtt{UPDATE\text{-}OK}$ Creation successful.

Close virtual channel for I

// React to closure of γ in round t_0^I for some c_P, c_Q s.t. $c_P + c_Q = \gamma.\mathsf{cash}$
Let γ_P be the sub-channel $\gamma.\mathsf{subchan}(P)$ for $P \in \{A, B\}$.
Compute $\vec{\theta}_P = \{(c_P, \mathtt{One\text{-}Sig}_{pk_P}), (c_Q + \frac{\gamma.\mathsf{fee}}{2}, \mathtt{One\text{-}Sig}_{pk_I})\}$
for $P \in \{A, B\}$.
if $\mathsf{UpdateChanSync}^I(\gamma_A, \vec{\theta}_A, \gamma_B, \vec{\theta}_B, \bot)$ returns $\mathtt{UPDATE\text{-}OK}$
Close successful.
else Execute $\mathtt{Offload}^I(\gamma)$.

Punish for all parties

// In every round check the ledger and punish misbehavior
for every open channel γ execute $\mathsf{Punish}(\gamma)$

$\mathsf{UpdateChan}^P(\gamma, \vec{\theta}, \mathsf{Setup})$ from [3]

// Initiate update of γ with state $\vec{\theta}$ in τ_0 with setup procedure Setup.

$\mathsf{UpdateChanSync}^I(\gamma_1, \vec{\theta}_1, \gamma_2, \vec{\theta}_2, \mathsf{Setup})$

// Initiate update of γ_i with state $\vec{\theta}_i$ with Setup for $i = 1$ and 2 simultaneously
// using the same steps of $\mathsf{UpdateChan}$. At each step, wait for both channels
// before continuing. If one of them fails at any step, act as both failed.

$\mathsf{PreCreateChan}(\mathtt{TX}_f^\gamma)$ from [3]

// Creates a channel γ with initial versions of split and commit transactions.
// It follows the channel creation procedure given in [3], expect that
// the funding transaction is not published in the end.

Fig. 13: Protocol for virtual channels. The protocol utilizes the generalized channel protocols from [3]. Specifically, the channel update protocol UpdateChan is used in a black-box fashion while also defining a synchronized version called UpdateChanSync. Moreover, the channel creation protocol PreCreateChan is used with the difference of not publishing the channel funding transaction of the virtual channel. The gray parts of the protocol differ between our tow constructions with and without validity and are specified in the protocol pseudocode and description.

C. Two-Party Adaptor Signatures From Identification Schemes

This chapter corresponds to our published article in Practice and Theory of Public-Key Cryptography (PKC), 2021 [70] (https://doi.org/10.1007/978-3-030-75245-3_17). Our full version can be found in [69].

Two-Party Adaptor Signatures From Identification Schemes

Andreas Erwig[1], Sebastian Faust[1], Kristina Hostáková[2,†], Monosij Maitra[1,‡], and Siavash Riahi[1]

[1] Technische Universität Darmstadt, Germany
firstname.lastname@tu-darmstadt.de
[2] ETH Zürich, Switzerland
kristina.hostakova@inf.ethz.ch

Abstract. Adaptor signatures are a novel cryptographic primitive with important applications for cryptocurrencies. They have been used to construct second layer solutions such as payment channels or cross-currency swaps. The basic idea of an adaptor signature scheme is to tie the signing process to the revelation of a secret value in the sense that, much like a regular signature scheme, an adaptor signature scheme can authenticate messages, but simultaneously leaks a secret to certain parties. Recently, Aumayr et al. provide the first formalization of adaptor signature schemes, and present provably secure constructions from ECDSA and Schnorr signatures. Unfortunately, the formalization and constructions given in this work have two limitations: (1) current schemes are limited to ECDSA and Schnorr signatures, and no generic transformation for constructing adaptor signatures is known; (2) they do not offer support for aggregated two-party signing, which can significantly reduce the blockchain footprint in applications of adaptor signatures.
In this work, we address these two shortcomings. First, we show that signature schemes that are constructed from identification (ID) schemes, which additionally satisfy certain homomorphic properties, can generically be transformed into adaptor signature schemes. We further provide an impossibility result which proves that unique signature schemes (e.g., the BLS scheme) cannot be transformed into an adaptor signature scheme. In addition, we define two-party adaptor signature schemes with aggregatable public keys and show how to instantiate them via a generic transformation from ID-based signature schemes. Finally, we give instantiations of our generic transformations for the Schnorr, Katz-Wang and Guillou-Quisquater signature schemes.

1 Introduction

Blockchain technologies, envisioned first in 2009 [34], have spurred enormous interest by academia and industry. This technology puts forth a decentralized payment paradigm, where financial transactions are stored in a decentralized data structure – often referred to as the blockchain. The main cryptographic primitive used by blockchain systems is the one of digital signature schemes, which allow users to authenticate payment transactions. Various different flavors of digital signature schemes are used by blockchain systems, e.g., ring signatures [39] add privacy-preserving features to cryptocurrencies [40], while threshold signatures and multi-signatures are used for multi-factor authorization of transactions [18].

† Research partially conducted at Technische Universität Darmstadt, Germany.
‡ Research partially conducted at Indian Institute of Technology Madras, India.

Adaptor signatures (sometimes also referred to as scriptless scripts) are another important type of digital signature scheme introduced by the cryptocurrency community [37] and recently formalized by Aumayr et al. [2]. In a nutshell, adaptor signatures tie together authorization of a message and the leakage of a secret value. Namely, they allow a *signer* to produce a *pre-signature* under her secret key such that this pre-signature can be *adapted* into a valid signature by a *publisher* knowing a certain secret value. If the completed signature gets published, the signer is able to extract the embedded secret used by the publisher.

To demonstrate the concept of adaptor signatures, let us discuss the simple example of a preimage sale which serves as an important building block in many blockchain applications such as payment channels [6, 10, 38, 2], payment routing in payment channel networks (PCNs) [30, 13, 33] or atomic swaps [11, 21]. Assume that a seller offers to reveal a preimage of a hash value h in exchange for c coins from a concrete buyer. This is a classical instance of a fair exchange problem, which can be solved using the blockchain as follows. The buyer locks c coins in a transaction which can be spent by another transaction if it is authorized by the seller and contains a preimage of the hash value h.

While this solution implements the preimage sale, it has various drawbacks: (i) The only hash functions that can be used are the ones supported by the underlying blockchain. For example, the most popular blockchain-based cryptocurrency, Bitcoin, supports only SHA-1, SHA-256 and RIPEMD-160 [5]. This makes the above solution unsuitable for applications like privacy-preserving payment routing in PCNs [30, 13] that crucially rely on the preimage sale instantiated with a *homomorphic* hash function. (ii) The hash value has to be fixed at the beginning of the sale and cannot be changed later without a new transaction being posted on the blockchain. This is problematic in, e.g., generalized payment channels [2], where users utilize the ideas from the preimage sale to repeatedly update channel balances without any blockchain interaction. (iii) Finally, the blockchain script is non-standard as, in addition to a signature verification, it contains a hash preimage verification. This does not only make the transaction more expensive but also allows parties who are maintaining the blockchain (also known as *miners*) to censor transactions belonging to a preimage sale.

The concept of adaptor signatures allows us to implement a preimage sale in a way that overcomes most of the aforementioned drawbacks. The protocol works at a high level as follows. The buyer locks c coins in a transaction which can be spent by a transaction authorized by *both* the seller and the buyer. Thereafter, the buyer pre-signs a transaction spending the c coins with respect to the hash value h. If the seller knows a preimage of h, she can adapt the pre-signature of the buyer, attach her own signature and claim the c coins. The buyer can then extract a preimage from the adapted signature. Hence, parties are not restricted to the hash functions supported by the blockchain, i.e., drawback (i) is addressed. Moreover, the buyer can pre-sign the spending transaction with respect to multiple hash values which overcomes drawback (ii). However, the third drawback remains. While the usage of adaptor signatures avoids the hash preimage verification in the script, it adds a signature

verification (i.e., there are now 2 signature verifications in total) which makes this type of exchange easily distinguishable from a normal payment transaction. Hence, the sale remains rather expensive and censorship is not prevented.

The idea of *two-party* adaptor signatures is to replace the two signature verifications by one. The transaction implementing a preimage sale then has exactly the same format as a transaction simply transferring coins. As a result the price (in terms of fees paid to the miners) of the preimage sale transaction is the same as the price for a normal payment. Moreover, censorship is prevented as miners cannot distinguish the transactions belonging to the preimage sale from a standard payment transaction. Hence, point (iii) is fully addressed.

The idea of replacing two signatures by one has already appeared in the literature in the context of payment channels. Namely, Malavolta et al. [30] presented protocols for two-party threshold adaptor signatures based on Schnorr and ECDSA digital signatures. However, they did not present a standalone definition for the threshold primitive and hence security for these schemes has not been analyzed. Furthermore, the key generation of the existing threshold adaptor signature schemes is interactive which is undesirable. Last but not least, their constructions are tailored to Schnorr and ECDSA signature schemes and hence is not generic. From the above points, the following natural question arises:

Is it possible to define and instantiate two-party adaptor signature schemes with non-interactive key generation in a generic way?

1.1 Our contribution

Our main goal is to define two-party adaptor signatures and explore from which digital signature we can instantiate this new primitive. We proceed in three steps which we summarize below and depict in Fig. 1.

Step 1: From ID schemes to adaptor signatures. Our first goal is to determine if there exists a specific class of signature schemes which can be generically transformed into adaptor signatures. Given the existing Schnorr-based construction [37, 2], a natural choice is to explore signature schemes constructed in a similar fashion. To this end, we focus on signature schemes built from identification (ID) schemes using the Fiat-Shamir transform [25]. We show that ID-based signature schemes satisfying certain additional properties can be transformed to adaptor signature schemes generically. In addition to Schnorr signatures [41], this class includes Katz-Wang and Guillou-Quisquater signatures [24, 22]. As an additional result, we show that adaptor signatures *cannot* be built from unique signatures, ruling out constructions from, e.g., BLS signatures [9].

Our generic transformation of adaptor signatures from ID schemes has multiple benefits. Firstly, by instantiating it with the Guillou-Quisquater siganture scheme, we obtain the first RSA-based adaptor signature scheme. Secondly, since Katz-Wang signatures offers tight security (under the decisional Diffie-Hellman (DDH) assumption), and our generic transformation also achieves tight security, our result shows

how to construct adaptor signatures with a tight reduction to the underlying DDH assumption.

Step 2: From ID schemes to two-party signatures. Our second goal is to generically transform signature schemes built from ID schemes into two-party signature schemes with aggregatable public keys. Unlike threshold signatures, these signatures have non-interactive key generation. This means that parties can independently generate their key pairs and later collaboratively generate signatures that are valid under their *combined* public key. For our transformation, we require the signature scheme to satisfy certain aggregation properties which, as we show, are present in the three aforementioned signature schemes. While this transformation serves as a middle step towards our main goal of constructing two-party adaptor signatures, we believe it is of independent interest.

Step 3: From ID schemes to two-party adaptor signatures. Finally, we define two-party adaptor signature schemes with aggregatable public keys. In order to instantiate this novel cryptographic primitive, we use similar techniques as in step 1 where we "lifted" standard signature schemes to adaptor signature schemes. More precisely, we present a transformation turning a two-party signature scheme based on an ID scheme into a two-party adaptor signature scheme.

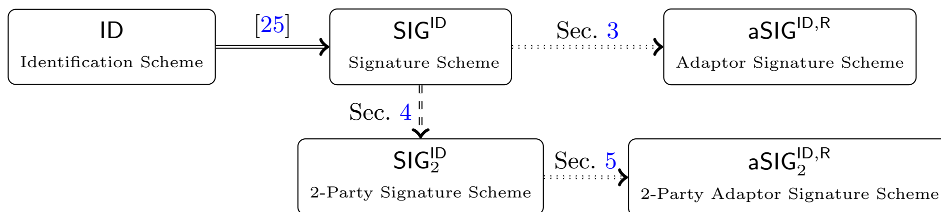

Fig. 1: Overview of our results. Full arrow represents a generic transformation, dotted and dashed arrows represent a generic transformation which requires additional homomorphic or aggregation properties respectively.

Remark 1. Let us point out that Fig. 1 presents our transformation steps from signature schemes based on ID schemes to two-party adaptor signatures. Despite the fact that we generically construct our two-party adaptor signature scheme from two-party signature schemes based on ID schemes, we reduce its security to the strong unforgeability of the underlying single party signature scheme. Therefore, we do not need the two-party signature scheme from ID schemes to be strongly unforgeable. This gives us a more general result than proving security based on strong unforgeability of the two-party signature scheme from ID schemes. We note that any ID scheme can be transformed to a signature scheme with strong unforgeability by Bellare and Shoup [4].

Let us further mention that our security proofs are in the random oracle model. Proving the security of our constructions and the original constructions from [2] in the standard model remains an interesting open problem.

1.2 Related Work

Adaptor Signatures. The notion of adaptor signatures was first introduced by Poelstra [37] and has since been used in many blockchain related applications, such as PCNs [30], payment channel hubs [43] or atomic swaps [11]. However, the adaptor signatures as a standalone primitive were only formalized later by Aumayr et al. [2], where they were used to generalize the concept of payment channels. Concurrently, Fournier [17] attempted to formalize adaptor signatures, however, as pointed out in [2], his definition is weaker than the one given in [2] and not sufficient for certain applications. All the previously mentioned works constructed adaptor signatures only from Schnorr and ECDSA signatures, i.e., they did not show generic transformations for building adaptor signature schemes. As previously mentioned, a two-party threshold variant of adaptor signatures was presented by Malavolta et al. [30]. Their construction requires interactive key generation, thereby differing from our two-party adaptor signature notion. Moreover, no standalone definition of the threshold primitive was provided.

Two works [15, 44] have recently introduced post-quantum secure adaptor signature schemes, i.e., schemes that remain secure even in presence of an adversary having access to a quantum computer. In order to achieve post-quantum security, [15] based its scheme on standard and well-studied lattice assumptions, namely Module-SIS and Module-LWE, while the scheme in [44] is based on lesser known assumptions for isogenies. Both works additionally show how to construct post-quantum secure PCNs from their respective adaptor signature schemes.

Multi-Signatures and ID Schemes. Multi-Signatures have been subject to extensive research in the past (e.g., [36, 35, 23]). In a nutshell, multi-signatures allow a set of signers to collaboratively generate a signature for a common message such that the signature can be verified given the public key of each signer. More recently, the notion of multi-signatures with aggregatable public keys has been introduced [31] and worked on [8, 26], which allows to aggregate the public keys of all signers into one single public key. We use some results from the work of Kiltz et al. [25], which provides a concrete and modular security analysis of signatures schemes from ID schemes obtained via the Fiat-Shamir transformation. Our paper builds up on their work and uses some of their notation.

2 Preliminaries

In this section, we introduce notation that we use throughout this work and preliminaries on adaptor signatures and identification schemes. Due to space limitations, we provide formal definitions of digital signature schemes, non-interactive zero-knowledge proofs and extractable commitments in the full version of this paper [14].

Notation. We denote by $x \leftarrow_\$ \mathcal{X}$ the uniform sampling of x from the set $\mathcal{X}$. Throughout this paper, n denotes the security parameter. By $x \leftarrow \mathsf{A}(y)$ we denote a *probabilistic polynomial time* (PPT) algorithm A that on input y, outputs x. When A is a *deterministic polynomial time* (DPT) algorithm, we use the notation $x := \mathsf{A}(y)$. A function $\nu\colon \mathbb{N} \to \mathbb{R}$ is *negligible in* n if for every $k \in \mathbb{N}$, there exists $n_0 \in \mathbb{N}$ s.t. for every $n \geq n_0$ it holds that $|\nu(n)| \leq 1/n^k$.

Hard relation. Let $\mathsf{R} \subseteq \mathcal{D}_S \times \mathcal{D}_\mathsf{w}$ be a relation with statement/witness pairs $(Y, y) \in \mathcal{D}_S \times \mathcal{D}_\mathsf{w}$ and let the language $L_\mathsf{R} \subseteq \mathcal{D}_S$ associated to R be defined as $L_\mathsf{R} := \{Y \in \mathcal{D}_S \mid \exists y \in \mathcal{D}_\mathsf{w} \text{ s.t. } (Y, y) \in \mathsf{R}\}$. We say that R is a *hard relation* if: (i) There exists a PPT sampling algorithm $\mathsf{GenR}(1^n)$ that on input the security parameter outputs a pair $(Y, y) \in \mathsf{R}$; (ii) The relation R is poly-time decidable; (iii) For all PPT adversaries $\mathcal{A}$, the probability that $\mathcal{A}$ outputs a valid witness $y \in \mathcal{D}_\mathsf{w}$ for $Y \in L_\mathsf{R}$ is negligible.

2.1 Adaptor Signatures

We now recall the definition of adaptor signatures, recently put forward in [2].

Definition 1 (Adaptor signature). *An adaptor signature scheme w.r.t. a hard relation* R *and a signature scheme* $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *consists of a tuple of four algorithms* $\mathsf{aSIG}_{\mathsf{R},\mathsf{SIG}} = (\mathsf{pSign}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ *defined as:*

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

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

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

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

An adaptor signature scheme, besides satisfying plain digital signature correctness, should also satisfy pre-signature correctness that we formalize next.

Definition 2 (Pre-signature correctness). *An adaptor signature* $\mathsf{aSIG}_{\mathsf{R},\mathsf{SIG}}$ *satisfies* pre-signature correctness*, if for all* $n \in \mathbb{N}$ *and* $m \in \{0,1\}^*$*:*

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

An adaptor signature scheme $\mathsf{aSIG}_{\mathsf{R},\mathsf{SIG}}$ is called *secure* if it satisfies three security properties: *existential unforgeablity under chosen message attack for adaptor signatures*, *pre-signature adaptability* and *witness extractability*. Let us recall the formal definition of these properties next.

The notion of unforgeability for adaptor signatures is similar to existential unforgeability under chosen message attacks for standard digital signatures but additionally requires that producing a forgery σ for some message m^* is hard even given a pre-signature on m^* w.r.t. a random statement $Y \in L_\mathsf{R}$.

Definition 3 (aEUF–CMA Security). *An adaptor signature scheme* $\mathsf{aSIG}_{\mathsf{R,SIG}}$ *is unforgeable if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* ν *such that:* $\Pr[\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_{\mathsf{R,SIG}}}(n) = 1] \leq \nu(n)$*, where the definition of the experiment* $\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_{\mathsf{R,SIG}}}$ *is as follows:*

$\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_{\mathsf{R,SIG}}}(n)$	$\mathcal{O}_{\mathrm{S}}(m)$	$\mathcal{O}_{\mathrm{pS}}(m, Y)$
$1: \mathcal{Q} := \emptyset, (sk, pk) \leftarrow \mathsf{Gen}(1^n)$	$1: \sigma \leftarrow \mathsf{Sign}_{sk}(m)$	$1: \widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$
$2: m^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}}}(pk)$	$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$	$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3: (Y, y) \leftarrow \mathsf{GenR}(1^n), \widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m^*, Y)$	$3:$ **return** σ	$3:$ **return** $\widetilde{\sigma}$
$4: \sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}}}(\widetilde{\sigma}, Y)$		
$5:$ **return** $\left(m^* \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{pk}(m^*; \sigma^*)\right)$		

A natural requirement for an adaptor signature scheme is that any valid pre-signature w.r.t. Y (possibly produced by a malicious signer) can be completed into a valid signature using a witness y with $(Y, y) \in \mathsf{R}$.

Definition 4 (Pre-signature adaptability). *An adaptor signature scheme* $\mathsf{aSIG}_{\mathsf{SIG,R}}$ *satisfies pre-signature adaptability, if for all* $n \in \mathbb{N}$*, messages* $m \in \{0,1\}^*$*, statement/witness pairs* $(Y, y) \in \mathsf{R}$*, public keys* pk *and pre-signatures* $\widetilde{\sigma} \leftarrow \{0,1\}^*$ *we have* $\mathsf{pVrfy}_{pk}(m, Y; \widetilde{\sigma}) = 1$*, then* $\mathsf{Vrfy}_{pk}(m; \mathsf{Adapt}_{pk}(\widetilde{\sigma}, y)) = 1$.

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

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

$\mathsf{aWitExt}_{\mathcal{A},\mathsf{aSIG}_{\mathsf{R,SIG}}}(n)$	$\mathcal{O}_{\mathrm{S}}(m)$	$\mathcal{O}_{\mathrm{pS}}(m, Y)$
$1: \mathcal{Q} := \emptyset, (sk, pk) \leftarrow \mathsf{Gen}(1^n)$	$1: \sigma \leftarrow \mathsf{Sign}_{sk}(m)$	$1: \widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$
$2: (m^*, Y^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}}}(pk)$	$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$	$2: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$3: \widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m^*, Y^*)$	$3:$ **return** σ	$3:$ **return** $\widetilde{\sigma}$
$4: \sigma^* \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{S}},\mathcal{O}_{\mathrm{pS}}}(\widetilde{\sigma})$		
$5: y := \mathsf{Ext}_{pk}(\sigma^*, \widetilde{\sigma}, Y^*)$		
$6:$ **return** $(m^* \notin \mathcal{Q} \wedge (Y^*, y) \notin \mathsf{R} \wedge \mathsf{Vrfy}_{pk}(m^*; \sigma^*))$		

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

2.2 Identification and Signature Schemes

In this section we recall the definition of identification schemes and how they are transformed to signature schemes as described in [25].

Definition 6 (Canonical Identification Scheme [25]). *A canonical identification scheme* ID *is defined as a tuple of four algorithms* $\mathsf{ID} := (\mathsf{IGen}, \mathsf{P}, \mathsf{ChSet}, \mathsf{V})$.

- *The key generation algorithm* IGen *takes the system parameters* par *as input and returns secret and public key* (sk, pk)*. We assume that* pk *defines the set of challenges, namely* ChSet.
- *The prover algorithm* P *consists of two algorithms namely* P_1 *and* P_2*:*
  - P_1 *takes as input the secret key* sk *and returns a commitment* $R \in \mathcal{D}_{\mathsf{rand}}$ *and a state* St.
  - P_2 *takes as input the secret key* sk*, a commitment* $R \in \mathcal{D}_{\mathsf{rand}}$*, a challenge* $h \in \mathsf{ChSet}$*, and a state* St *and returns a response* $s \in \mathcal{D}_{\mathsf{resp}}$.
- *The verifier algorithm* V *is a deterministic algorithm that takes the public key* pk *and the conversation transcript as input and outputs* 1 *(acceptance) or* 0 *(rejection).*

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

We recall that an identification scheme ID is called *commitment-recoverable*, if V first internally calls a function V_0 which recomputes $R_0 = \mathsf{V}_0(pk, h, s)$ and then outputs 1, iff $R_0 = R$. Using Fiat-Shamir heuristic one can transform any identification scheme ID of the above form into a digital signature scheme $\mathsf{SIG}^{\mathsf{ID}}$. We recall this transformation in Fig. 2 when ID is commitment-recoverable.

$\mathsf{Gen}(1^n)$	$\mathsf{Sign}_{sk}(m)$	$\mathsf{Vrfy}_{pk}(m; (h, s))$
$1: (sk, pk) \leftarrow \mathsf{IGen}(n)$	$1: (R, St) \leftarrow \mathsf{P}_1(sk)$	$1: R := \mathsf{V}_0(pk, h, s)$
$2: \textbf{return } (sk, pk)$	$2: h := \mathcal{H}(R, m)$	$2: \textbf{return } h = \mathcal{H}(R, m)$
	$3: s \leftarrow \mathsf{P}_2(sk, R, h, St)$	
	$4: \textbf{return } (h, s)$	

Fig. 2: $\mathsf{SIG}^{\mathsf{ID}}$: Digital signature schemes from identification schemes [25]

3 Adaptor Signatures from $\mathsf{SIG}^{\mathsf{ID}}$

Our first goal is to explore and find digital signature schemes which can generically be transformed to adaptor signatures. Interestingly, we observe that both existing adaptor signature schemes, namely the Schnorr-based and the ECDSA-based schemes, utilize the randomness used during signature generation to transform digital signatures to adaptor signatures [2]. We first prove a negative result, namely

that it is impossible to construct an adaptor signature scheme from a unique signature scheme [42, 29, 19]. Thereafter, we focus on signature schemes constructed from identification schemes (cf. Fig. 2) and show that if the underlying ID-based signature scheme $\mathsf{SIG}^{\mathsf{ID}}$ satisfies certain additional properties, then we can generically transform it into an adaptor signature scheme. To demonstrate the applicability of our generic transformation, we show in the full version of this paper [14] that many existing $\mathsf{SIG}^{\mathsf{ID}}$ instantiations satisfy the required properties.

3.1 Impossibility Result for Unique Signatures

An important class of digital signatures are those where the signing algorithm is deterministic and the generated signatures are unique. Given the efficiency of deterministic signature schemes along with numerous other advantages that come from signatures being unique [42, 29, 19], it would be tempting to design adaptor signatures based on unique signatures. However, we show in Thm. 1 that if the signature scheme has unique signatures, then it is impossible to construct a secure adaptor signature scheme from it.

Theorem 1. *Let* R *be a hard relation and* $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ *be a signature scheme with unique signatures. Then there does not exist an adaptor signature scheme* $\mathsf{aSIG}_{\mathsf{R},\mathsf{SIG}}$.

Proof. We prove this theorem by contradiction. Assume there exists an adaptor signature scheme where the underlying signature scheme, SIG, has unique signatures. We construct a PPT algorithm $\mathcal{A}$ which internally uses the adaptor signature and breaks the hardness of R. In other words, $\mathcal{A}$ receives $(1^n, Y)$ as input and outputs y, such that $(Y, y) \in \mathsf{R}$. Below, we describe $\mathcal{A}$ formally.

On input $(1^n, Y)$, $\mathcal{A}$ proceeds as follows:

1 : Sample a new key pair $(sk, pk) \leftarrow \mathsf{Gen}(1^n)$.
2 : Choose an arbitrary message m from the signing message space.
3 : Generate a pre-signature, $\widetilde{\sigma} \leftarrow \mathsf{preSign}_{sk}(m, Y)$.
4 : Generate a signature, $\sigma := \mathsf{Sign}_{sk}(m)$.
5 : Compute and output $y := \mathsf{Ext}_{pk}(\sigma, \widetilde{\sigma}, Y)$.

We now show that y returned by $\mathcal{A}$ is indeed a witness of Y, i.e., $(Y, y) \in \mathsf{R}$. From the correctness of the adaptor signature scheme, we know that for any y' s.t. $(Y, y') \in \mathsf{R}$ the signature $\sigma' := \mathsf{Adapt}(\widetilde{\sigma}, y')$ is a valid signature, i.e., $\mathsf{Vrfy}_{pk}(m, \sigma') = 1$. Moreover, we know that $y'' := \mathsf{Ext}_{pk}(\sigma', \widetilde{\sigma}, Y)$ is such that $(Y, y'') \in \mathsf{R}$. As SIG is a unique signature scheme, this implies that $\sigma' = \sigma$ which in turn implies that the witness y returned by $\mathcal{A}$ is y''. Hence, $\mathcal{A}$ breaks the hardness of R with probability 1.

Let us briefly discuss which signature schemes are affected by our impossibility result. Unique signature schemes (also known as verifiable unpredictable functions (VUF)) have been first introduced in [19]. Furthermore, many follow-up works such

as [32, 29] and most recently [42], have shown how to instantiate this primitive in the standard model. Another famous example of a unique signature scheme is BLS [9]. Naturally, due to our impossibility result, an adaptor signature scheme cannot be instantiated from these signature schemes.

3.2 Generic Transformation to Adaptor Signatures

We now describe how to generically transform a randomized digital signature scheme $\mathsf{SIG}^{\mathsf{ID}}$ from Fig. 2 into an adaptor signature scheme w.r.t. a hard relation R. For brevity, we denote the resulting adaptor signature scheme as $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ instead of $\mathsf{aSIG}_{\mathsf{R},\mathsf{SIG}^{\mathsf{ID}}}$. The main idea behind our transformation is to *shift* the public randomness of the Sign procedure by a statement Y for the relation R in order to generate a modified signature called a *pre-signature*. Using a corresponding witness y (i.e., $(Y, y) \in \mathsf{R}$), the shift of the public randomness in the pre-signature can be reversed (or adapted), in order to obtain a regular (or full) signature. Moreover, it should be possible to extract a witness given both the pre-signature and the full-signature. To this end, let us formalize three new *deterministic* functions which we will use later in our transformation.

1. For the randomness shift, we define a function $f_{\mathsf{shift}} \colon \mathcal{D}_{\mathsf{rand}} \times L_{\mathsf{R}} \to \mathcal{D}_{\mathsf{rand}}$ that takes as input a commitment value $R \in \mathcal{D}_{\mathsf{rand}}$ of the identification scheme and a statement $Y \in L_{\mathsf{R}}$ of the hard relation, and outputs a new commitment value $R' \in \mathcal{D}_{\mathsf{rand}}$.
2. For the adapt operation, we define $f_{\mathsf{adapt}} \colon \mathcal{D}_{\mathsf{resp}} \times \mathcal{D}_{\mathsf{w}} \to \mathcal{D}_{\mathsf{resp}}$ that takes as input a response value $\tilde{s} \in \mathcal{D}_{\mathsf{resp}}$ of the identification scheme and a witness $y \in \mathcal{D}_{\mathsf{w}}$ of the hard relation, and outputs a new response value $s \in \mathcal{D}_{\mathsf{resp}}$.
3. Finally, for witness extraction, we define $f_{\mathsf{ext}} \colon \mathcal{D}_{\mathsf{resp}} \times \mathcal{D}_{\mathsf{resp}} \to \mathcal{D}_{\mathsf{w}}$ that takes as input two response values $\tilde{s}, s \in \mathcal{D}_{\mathsf{resp}}$ and outputs a witness $y \in \mathcal{D}_{\mathsf{w}}$.

Our transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ is shown in Fig. 3.

$\mathsf{pSign}_{sk}(m, Y)$	$\mathsf{pVrfy}_{pk}(m, Y; (h, \tilde{s}))$	$\mathsf{Adapt}_{pk}((h, \tilde{s}), y)$
$1: (R_{\mathsf{pre}}, St) \leftarrow \mathsf{P}_1(sk)$	$1: \widehat{R}_{\mathsf{pre}} := \mathsf{V}_0(pk, h, \tilde{s})$	$1: s = f_{\mathsf{adapt}}(\tilde{s}, y)$
$2: R_{\mathsf{sign}} := f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y)$	$2: \widehat{R}_{\mathsf{sign}} := f_{\mathsf{shift}}(\widehat{R}_{\mathsf{pre}}, Y)$	$2: \textbf{return } (h, s)$
$3: h := \mathcal{H}(R_{\mathsf{sign}}, m)$	$3: b := (h = \mathcal{H}(\widehat{R}_{\mathsf{sign}}, m))$	$\mathsf{Ext}_{pk}((h, s), (h, \tilde{s}), Y)$
$4: \tilde{s} \leftarrow \mathsf{P}_2(sk, R_{\mathsf{pre}}, h, St)$	$4: \textbf{return } b$	$1: \textbf{return } f_{\mathsf{ext}}(s, \tilde{s})$
$5: \textbf{return } (h, \tilde{s})$		

Fig. 3: Generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to a $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ scheme

In order for $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ to be an adaptor signature scheme, we need the functions f_{shift}, f_{adapt} and f_{ext} to satisfy two properties. The first property is a homomorphic one and relates the functions f_{shift} and f_{adapt} to the commitment-recoverable component

$\mathsf{IGen}(n)$	$\mathsf{P}_1(sk)$	$\mathsf{P}_2(sk, R, h, r)$	$\mathsf{V}_0(pk, h, s)$
$1: sk \leftarrow_\$ \mathbb{Z}_q, pk = g^{sk}$	$1: r \leftarrow_\$ \mathbb{Z}_q, R = g^r$	$1: s = r + h \cdot sk$	$1: R = g^s \cdot pk^{-h}$
$2:$ **return** (sk, pk)	$2:$ **return** (R, r)	$2:$ **return** s	$2:$ **return** (R)

Fig. 4: Schnorr signature scheme

V_0 and the hard relation R. Informally, for all $(Y, y) \in \mathsf{R}$, we need the following to be equivalent: (i) Extract the public randomness from a response $\tilde{s}$ using V_0 and then apply f_{shift} to shift the public randomness by Y, and (ii) apply f_{adapt} to shift the *secret* randomness in $\tilde{s}$ by y and then extract the public randomness using V_0. Formally, for any public key pk, any challenge $h \in \mathsf{ChSet}$, any response value $\tilde{s} \in \mathcal{D}_{\mathsf{resp}}$ and any statement/witness pair $(Y, y) \in \mathsf{R}$, it must hold that:

$$f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, \tilde{s}), Y) = \mathsf{V}_0(pk, h, f_{\mathsf{adapt}}(\tilde{s}, y)). \tag{1}$$

The second property requires that the function $f_{\mathsf{ext}}(\tilde{s}, \cdot)$ is the inverse function of $f_{\mathsf{adapt}}(\tilde{s}, \cdot)$ for any $\tilde{s} \in \mathcal{D}_{\mathsf{resp}}$. Formally, for any $y \in \mathcal{D}_{\mathsf{w}}$ and $\tilde{s} \in \mathcal{D}_{\mathsf{resp}}$, we have

$$y = f_{\mathsf{ext}}(f_{\mathsf{adapt}}(\tilde{s}, y), \tilde{s}). \tag{2}$$

To give an intuition about the functions f_{shift}, f_{adapt} and f_{ext} and their purpose, let us discuss their concrete instantiations for Schnorr signatures and show that they satisfy Equations (1) and (2). The instantiations for Katz-Wang signatures and Guillou-Quisquater signatures can be found in the full version of this paper [14].

Example 1 (Schnorr signatures). Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order p where the discrete logarithm problem in $\mathbb{G}$ is hard. The functions IGen, P_1, P_2 and V_0 for Schnorr's signature scheme are defined in Fig. 4.

Let us consider the hard relation $\mathsf{R} = \{(Y, y) \mid Y = g^y\}$, i.e., group elements and their discrete logarithms, and let us define the functions f_{shift}, f_{adapt}, f_{ext} as:

$$f_{\mathsf{shift}}(Y, R) := Y \cdot R, \quad f_{\mathsf{adapt}}(\tilde{s}, y) := \tilde{s} + y, \quad f_{\mathsf{ext}}(s, \tilde{s}) := s - \tilde{s}.$$

Intuitively, the function f_{shift} is *shifting* randomness in the group while the function f_{adapt} *shifts* randomness in the exponent. To prove that Eq. (1) holds, let us fix an arbitrary public key $pk \in \mathbb{G}$, a challenge $h \in \mathbb{Z}_q$, a response value $s \in \mathbb{Z}_q$ and a statement witness pair $(Y, y) \in \mathsf{R}$, i.e, $Y = g^y$. We have

$$\begin{aligned} f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, s), Y) &= f_{\mathsf{shift}}(g^s \cdot pk^{-h}, Y) = g^s \cdot pk^{-h} \cdot Y \\ &= g^{s+y} \cdot pk^{-h} = \mathsf{V}_0(pk, h, s + y) = \mathsf{V}_0(pk, h, f_{\mathsf{adapt}}(s, y)) \end{aligned}$$

which is what we wanted to prove. In order to show that Eq. (2) holds, let us fix an arbitrary witness $y \in \mathbb{Z}_q$ and a response value $s \in \mathbb{Z}_q$. Then we have

$$f_{\mathsf{ext}}(f_{\mathsf{adapt}}(s, y), s) = f_{\mathsf{ext}}(s + y, s) = s + y - s = y$$

and hence Eq. (2) is satisfied as well.

We now show that the transformation from Fig. 3 is a secure adaptor signature scheme if functions $f_{\mathsf{shift}}, f_{\mathsf{adapt}}, f_{\mathsf{ext}}$ satisfying Equations (1) and (2) exist.

Theorem 2. *Assume that* $\mathsf{SIG}^{\mathsf{ID}}$ *is a* $\mathsf{SUF{-}CMA}$*-secure signature scheme transformed using Fig. 2, let* $f_{\mathsf{shift}}, f_{\mathsf{adapt}}$ *and* f_{ext} *be functions satisfying the relations from Equations (1) and (2), and* R *be a hard relation. Then the resulting* $\mathsf{aSIG}^{\mathsf{ID,R}}$ *scheme from the transformation in Fig. 3 is a secure adaptor signature scheme in the random oracle model.*

In order to prove Thm. 2, we must show that $\mathsf{aSIG}^{\mathsf{ID,R}}$ satisfies *pre-signature correctness*, $\mathsf{aEUF{-}CMA}$ *security*, *pre-signature adaptability* and *witness extractability* properties described in Defs. 2 to 5 respectively.

Lemma 1 (Pre-Signature Correctness). *Under the assumptions of Thm. 2,* $\mathsf{aSIG}^{\mathsf{ID,R}}$ *satisfies pre-signature correctness as for Def. 2.*

Proof. Let us fix an arbitrary message m and a statement witness pair $(Y, y) \in \mathsf{R}$. Let $(sk, pk) \leftarrow \mathsf{Gen}(1^n)$, $\widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m, Y)$, $\sigma := \mathsf{Adapt}_{pk}(\widetilde{\sigma}, y)$ and $y' := \mathsf{Ext}_{pk}(\sigma, \widetilde{\sigma}, Y)$. From Fig. 3 we know that $\widetilde{\sigma} = (h, \tilde{s})$, $\sigma = (h, s)$ and $y' = f_{\mathsf{ext}}(s, \tilde{s})$, where we have $s := f_{\mathsf{adapt}}(\tilde{s}, y)$, $\tilde{s} \leftarrow \mathsf{P}_2(sk, R_{\mathsf{pre}}, h, St)$, $h := \mathcal{H}(R_{\mathsf{sign}}, m)$, $R_{\mathsf{sign}} := f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y)$ and $(R_{\mathsf{pre}}, St) \leftarrow \mathsf{P}_1(sk)$. We first show $\mathsf{pVrfy}_{pk}(m, Y; \widetilde{\sigma}) = 1$. From completeness of the ID scheme, we know that $\mathsf{V}_0(pk, h, \tilde{s}) = R_{\mathsf{pre}}$. Hence:

$$\mathcal{H}(f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, \tilde{s}), Y), m) = \mathcal{H}(f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y), m) = \mathcal{H}(R_{\mathsf{sign}}, m) = h \tag{3}$$

which is what we needed to prove. We now show that $\mathsf{Vrfy}_{pk}(m; \sigma) = 1$. By Fig. 2, we need to show that $h = \mathcal{H}(\mathsf{V}_0(pk, h, s), m)$. This follows from the property of f_{shift}, f_{adapt} (cf. Eq. (1)) and Eq. (3) as follows:

$$\begin{aligned}\mathcal{H}(\mathsf{V}_0(pk, h, s), m) =& \mathcal{H}(\mathsf{V}_0(pk, h, f_{\mathsf{adapt}}(\tilde{s}, y)), m) \\ &\overset{(1)}{=} \mathcal{H}(f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, \tilde{s}), Y), m) \overset{(3)}{=} h.\end{aligned}$$

Finally, we need to show that $(Y, y') \in \mathsf{R}$. This follows from Eq. (2) since:

$$y' = f_{\mathsf{ext}}(s, \tilde{s}) = f_{\mathsf{ext}}(f_{\mathsf{adapt}}(\tilde{s}, y), \tilde{s}) \overset{(2)}{=} y.$$

Lemma 2 ($\mathsf{aEUF{-}CMA}$-Security). *Under the assumptions of Thm. 2,* $\mathsf{aSIG}^{\mathsf{ID,R}}$ *satisfies the* $\mathsf{aEUF{-}CMA}$ *security as for Def. 3.*

Let us give first a high level overview of the proof. Our goal is to provide a reduction such that, given an adversary $\mathcal{A}$ who can win the experiment $\mathsf{aSigForge}_{\mathcal{A}, \mathsf{aSIG}^{\mathsf{ID,R}}}$, we can build a simulator who can win the $\mathsf{strongSigForge}$ experiment of the underlying signature or can break the hardness of the relation R. In the first case, we check if $\mathcal{A}$'s forgery σ^* is equal to $\mathsf{Adapt}_{pk}(\widetilde{\sigma}, y)$. If so, we use $\mathcal{A}$ to break the hardness of the relation R by extracting the witness $y = \mathsf{Ext}(\sigma^*, \widetilde{\sigma}, Y)$. Otherwise, $\mathcal{A}$ was able to forge a signature "unrelated" to the pre-signature provided to it. In this case, it is used to win the $\mathsf{strongSigForge}$ experiment. All that remains is to answer $\mathcal{A}$'s signing and pre-signing queries using $\mathsf{strongSigForge}$'s signing queries. This is done by programming the random oracle such that the full-signatures generated by the challenger in the $\mathsf{strongSigForge}$ game look like pre-signatures for $\mathcal{A}$.

Proof. We prove the lemma by defining a series of game hops. The modifications for each game hop is presented in code form in the full version of this paper [14].

Game G_0: This game is the original $\mathsf{aSigForge}$ experiment, where the adversary $\mathcal{A}$ outputs a valid forgery σ^* for a message m of its choice, while having access to pre-signing and signing oracles $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$ respectively. Being in the random oracle model, all the algorithms of the scheme and the adversary have access to the random oracle $\mathcal{H}$. Since G_0 corresponds to $\mathsf{aSigForge}$, it follows that $\Pr[\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}}(n) = 1] = \Pr[G_0 = 1]$.
Game G_1: This game works as G_0 except when the adversary outputs a forgery σ^*, the game checks if adapting the pre-signature $\widetilde{\sigma}$ using the secret witness y results in σ^*. If so, the game aborts.

Claim. Let Bad_1 be the event where G_1 aborts. Then $\Pr[\mathsf{Bad}_1] \leq \nu_1(n)$, where ν_1 is a negligible function in n.

Proof: This claim is proven by a reduction to the relation R. We construct a simulator $\mathcal{S}$ which breaks the hardness of R using $\mathcal{A}$ that causes G_1 to abort with non-negligible probability. The simulator receives a challenge Y^*, and generates a key pair $(sk, pk) \leftarrow \mathsf{Gen}(1^n)$ in order to simulate $\mathcal{A}$'s queries to the oracles $\mathcal{H}$, $\mathcal{O}_{\mathrm{pS}}$ and $\mathcal{O}_{\mathrm{S}}$. This simulation of the oracles work as described in G_1.

Upon receiving the challenge message m^* from $\mathcal{A}$, $\mathcal{S}$ computes a pre-signature $\widetilde{\sigma} \leftarrow \mathsf{pSign}_{sk}(m^*, Y^*)$ and returns the pair $(\widetilde{\sigma}, Y)$ to the adversary. Upon $\mathcal{A}$ outputting a forgery σ^* and assuming that Bad_1 happened (i.e., $\mathsf{Adapt}(\widetilde{\sigma}, y) = \sigma$), pre-signature correctness (Def. 2) implies that the simulator can extract y^* by executing $\mathsf{Ext}(\sigma^*, \widetilde{\sigma}, Y^*)$ in order to obtain $(Y^*, y^*) \in \mathsf{R}$.

We note that the view of $\mathcal{A}$ in this simulation and in G_1 are indistinguishable, since the challenge Y^* is an instance of the hard relation R and has the same distribution to the public output of GenR. Therefore, the probability that $\mathcal{S}$ breaks the hardness of R is equal to the probability that the event Bad_1 happens. Hence, we conclude that Bad_1 only happens with negligible probability. ■

Since games G_1 and G_0 are equivalent except if event Bad_1 occurs, it holds that $\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \nu_1(n)$.

Game G_2: This game is similar to the previous game except for a modification in the $\mathcal{O}_{\mathrm{pS}}$ oracle. After the execution of $\mathsf{preSign}_{sk}$, the oracle obtains a pre-signature $\widetilde{\sigma}$ from which it extracts the randomness $R_{\mathsf{pre}} \leftarrow \mathsf{V}_0(pk, \widetilde{\sigma})$. The oracle computes $R_{\mathsf{sign}} = f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y)$ and checks if $\mathcal{H}$ was already queried on the inputs $R_{\mathsf{pre}} \| m$ or $R_{\mathsf{sign}} \| m$ *before* the execution of pSign_{sk}. In this case the game aborts.

Claim. Let Bad_2 be the event that G_2 aborts in $\mathcal{O}_{\mathrm{pS}}$. Then $\Pr[\mathsf{Bad}_2] \leq \nu_2(n)$, where ν_2 is a negligible function in n.

Proof: We first recall that the output of P_1 (i.e., R_{pre}) is uniformly random from a super-polynomial set of size q in the security parameter. From this it follows that R_{sign} is distributed uniformly at random in the same set. Furthermore, $\mathcal{A}$ being a PPT algorithm, it can only make polynomially many queries to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and $\mathcal{O}_{\mathrm{pS}}$

oracles. Denoting ℓ as the total number of queries to $\mathcal{H}$, $\mathcal{O}_{\mathrm{S}}$ and $\mathcal{O}_{\mathrm{pS}}$, we have: $\Pr[\mathsf{Bad}_2] = \Pr[H'[R_{\mathsf{pre}}||m] \neq \perp \vee H'[R_{\mathsf{sign}}||m] \neq \perp] \leq 2\frac{\ell}{q} \leq \nu_2(n)$. This follows from the fact that ℓ is polynomial in the security parameter. ■

Since games $\boldsymbol{G_2}$ and $\boldsymbol{G_1}$ are identical except in the case where Bad_2 occurs, it holds that $\Pr[\boldsymbol{G_1} = 1] \leq \Pr[\boldsymbol{G_2} = 1] + \nu_2(n)$.

Game $\boldsymbol{G_3}$: In this game, upon a query to the $\mathcal{O}_{\mathrm{pS}}$, the game produces a full-signature instead of a pre-signature by executing Sign_{sk} instead of $\mathsf{preSign}_{sk}$. Accordingly, it programs the random oracle $\mathcal{H}$ to make the full-signature "look like" a pre-signature from the point of view of the adversary $\mathcal{A}$. This is done by:

1. It sets $\mathcal{H}(R_{\mathsf{pre}}\|m)$ to the value stored at position $\mathcal{H}(R_{\mathsf{sign}}\|m)$.
2. It sets $\mathcal{H}(R_{\mathsf{sign}}\|m)$ to a fresh value chosen uniformly at random.

The above programming makes sense as our definition of f_{shift} requires it to be deterministic and to possess the same domain and codomain with respect to the commitment set $\mathcal{D}_{\mathsf{rand}}$. Note further that $\mathcal{A}$ can only notice that $\mathcal{H}$ was programmed if it was previously queried on either $R_{\mathsf{pre}}\|m$ or $R_{\mathsf{sign}}\|m$. But as described in the previous game, we abort if such an event happens. Hence, we have that $\Pr[\boldsymbol{G_2} = 1] = \Pr[\boldsymbol{G_3} = 1]$.

Game $\boldsymbol{G_4}$: In this game, we impose new checks during the challenge phase that are same as the ones imposed in $\boldsymbol{G_2}$ during the execution of $\mathcal{O}_{\mathrm{pS}}$.

Claim. Let Bad_3 be the event that $\boldsymbol{G_4}$ aborts in the challenge phase. Then $\Pr[\mathsf{Bad}_3] \leq \nu_3(n)$, where ν_3 is a negligible function in n.

Proof: The proof is identical to the proof in $\boldsymbol{G_2}$. ■

It follows that $\Pr[\boldsymbol{G_4} = 1] \leq \Pr[\boldsymbol{G_3} = 1] + \nu_3(n)$.

Game $\boldsymbol{G_5}$: Similar to game $\boldsymbol{G_3}$, we generate a signature instead of a pre-signature in the challenge phase and program $\mathcal{H}$ such that the full-signature looks like a correct pre-signature from $\mathcal{A}$'s point of view. We get $\Pr[\boldsymbol{G_5} = 1] = \Pr[\boldsymbol{G_4} = 1]$.

Now that the transition from the original $\mathsf{aSigForge}$ experiment (game $\boldsymbol{G_0}$) to game $\boldsymbol{G_5}$ is indistinguishable, it only remains to show the existence of a simulator $\mathcal{S}$ that can perfectly simulate $\boldsymbol{G_5}$ and uses $\mathcal{A}$ to win the $\mathsf{strongSigForge}$ game. The modifications from games $\boldsymbol{G_1}$ - $\boldsymbol{G_5}$ and the simulation in code form can be found in the full version of this paper [14].

We emphasize that the main differences between the simulation and Game $\boldsymbol{G_5}$ are syntactical. Namely, instead of generating the public and secret keys and computing the algorithm Sign_{sk} and the random oracle $\mathcal{H}$, $\mathcal{S}$ uses its oracles $\mathsf{SIG}^{\mathsf{ID}}$ and $\mathcal{H}^{\mathsf{ID}}$. Therefore, $\mathcal{S}$ perfectly simulates $\boldsymbol{G_5}$. It remains to show that $\mathcal{S}$ can use the forgery output by $\mathcal{A}$ to win the $\mathsf{strongSigForge}$ game.

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

Proof: To prove this claim, we show that the tuple (m^*, σ^*) has not been returned by the oracle $\mathsf{SIG}^{\mathsf{ID}}$ before. First note that $\mathcal{A}$ wins the experiment if it has not queried on the challenge message m^* to $\mathcal{O}_{\mathrm{pS}}$ or $\mathcal{O}_{\mathrm{S}}$. Therefore, $\mathsf{SIG}^{\mathsf{ID}}$ is queried on m^* only

during the challenge phase. If $\mathcal{A}$ outputs a forgery σ^* that is equal to the signature σ as output by $\mathsf{SIG}^{\mathsf{ID}}$, it would lose the game since this signature is not valid given the fact that $\mathcal{H}$ is programmed.

Hence, $\mathsf{SIG}^{\mathsf{ID}}$ has never output σ^* when queried on m^* before, thus making (m^*, σ^*) a valid forgery for game $\mathsf{strongSigForge}$. ■

From games $\boldsymbol{G_0} - \boldsymbol{G_5}$, we have that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_5} = 1] + \nu(n)$, where $\nu(n) = \nu_1(n) + \nu_2(n) + \nu_3(n)$ is a negligible function in n. Since $\mathcal{S}$ simulates game $\boldsymbol{G_5}$ perfectly, we also have that $\Pr[\boldsymbol{G_5} = 1] = \Pr[\mathsf{strongSigForge}_{\mathcal{S}^{\mathcal{A}},\mathsf{SIG}}(n) = 1]$. Combining this with the probability statement in $\boldsymbol{G_0}$, we obtain the following:

$\Pr[\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}}(n) = 1] \leq \Pr[\mathsf{strongSigForge}_{\mathcal{S}^{\mathcal{A}},\mathsf{SIG}^{\mathsf{ID}}}(n) = 1] + \nu(n)$.

Recall that the negligible function $\nu_1(n)$, contained in the sum $\nu(n)$ above, precisely quantifies the adversary's advantage in breaking the hard relation R. Thus, the probability of breaking the unforgeability of the $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ is clearly bounded above by that of breaking either R or the strong unforgeability of $\mathsf{SIG}^{\mathsf{ID}}$.

Lemma 3 (Pre-Signature Adaptability). *Under the assumptions of Thm. 2,* $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ *satisfies the pre-signature adaptability as for Def. 4.*

Proof. Assume $\mathsf{pVrfy}_{pk}(m, Y; \widetilde{\sigma}) = 1$, with the notations having their usual meanings from Fig. 3, which means $h = \mathcal{H}(f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, \tilde{s}), Y), m)$. For any valid pair $(Y, y) \in R$, we can use the homomorphic property from Eq. (1). Then, for such a pair $(Y, y) \in R$, plugging $f_{\mathsf{shift}}(\mathsf{V}_0(pk, h, \tilde{s}), Y) = \mathsf{V}_0(pk, h, f_{\mathsf{adapt}}(\tilde{s}, y))$ in the above equation implies $h = \mathcal{H}(\mathsf{V}_0(pk, h, f_{\mathsf{adapt}}(\tilde{s}, y)), m)$. This directly implies $\mathsf{Vrfy}_{pk}(m; \sigma) = 1$, where $s = f_{\mathsf{adapt}}(\tilde{s}, y)$ and $\sigma = (h, s)$. Therefore, adapting the valid pre-signature would also result in a valid full-signature.

Lemma 4 (Witness Extractability). *Under the assumptions of Thm. 2,* $\mathsf{aSIG}^{\mathsf{ID},\mathsf{R}}$ *satisfies the witness extractability as for Def. 5.*

This proof is very similar to the proof of Lemma 2 with the mere difference that we only need to provide a reduction to the $\mathsf{strongSigForge}$ experiment. This is because in the $\mathsf{aWitExt}_{\mathcal{A},\mathsf{aSIG}_{\mathsf{R}_g,\mathsf{SIG}^{\mathsf{ID}}}}$ experiment, $\mathcal{A}$ provides the public value Y^* and must forge a valid full-signature σ^* such that $(Y^*, \mathsf{Ext}_{pk}(\sigma^*, \widetilde{\sigma}, Y^*)) \notin \mathsf{R}$. The full proof can be found in the full version of this paper [14].

Remark 2. We note that our proofs for the $\mathsf{aEUF{-}CMA}$ security and witness extractability are in its essence reductions to the strong unforgeability of the underlying signature schemes. Yet the Fiat-Shamir transformation does not immediately guarantee the resulting signature scheme to be strongly unforgeable. However, we first note that many such signature schemes are indeed strongly unforgeable, for instance Schnorr [25], Katz-Wang (from Chaum-Pedersen identification scheme) [24] and Guillou-Quisquater [1] signature schemes all satisfy strong unforgeability. Moreover, one can transform any Fiat-Shamir based existentially unforgeable signature scheme into a strongly unforgeable one via the generic transformation using the results of Bellare et.al. [4].

4 Two-party Signatures with Aggregatable Public Keys from Identification Schemes

Before providing our definition and generic transformation for two-party adaptor signatures, we show how to generically transform signature schemes based on identification schemes into two-party signature schemes with aggregatable public keys denoted by SIG_2. In Sec. 5, we then combine the techniques used in this section with the ones from Sec. 3 in order to generically transform identification schemes into two-party adaptor signature schemes.

Informally, a SIG_2 scheme allows two parties to jointly generate a signature which can be verified under their combined public keys. An application of such signature schemes can be found in cryptocurrencies where two parties wish to only allow conditional payments such that both users have to sign a transaction in order to spend some funds. Using SIG_2, instead of submitting two separate signatures, the parties can submit a single signature while enforcing the same condition (i.e., a transaction must have a valid signature under the combined key) and hence reduce the communication necessary with the blockchain. Importantly and unlike threshold signature schemes, the key generation here is non-interactive. In other words, parties generate their public and secret keys independently and anyone who knows both public keys can compute the joint public key of the two parties.

We use the notation $\Pi_{\mathsf{Func}\langle x_i, x_{1-i}\rangle}$ to represent a two-party interactive protocol Func between P_i and P_{1-i} with respective secret inputs x_i, x_{1-i} for $i \in \{0,1\}$. Furthermore, if there are common public inputs e.g., $y_1, \cdots, y_n$ we use the notation $\Pi_{\mathsf{Func}\langle x_i, x_{1-i}\rangle}(y_1, \cdots, y_n)$. We note that the execution of a protocol might not be symmetric, i.e., party $\mathcal{P}_i$ executes the procedures $\Pi_{\mathsf{Func}\langle x_i, x_{1-i}\rangle}$ while party $\mathcal{P}_{1-i}$ executes the procedures $\Pi_{\mathsf{Func}\langle x_{1-i}, x_i\rangle}$.

4.1 Two-party Signatures with Aggregatable Public Keys

We start with defining a two-party signature scheme with aggregatable public keys. Our definition is inspired by the definitions from prior works [8, 26, 7].

Definition 7 (Two-party Signature with Aggregatable Public Keys). *A two-party signature scheme with aggregatable public keys is a tuple of PPT protocols and algorithms* $\mathsf{SIG}_2 = (\mathsf{Setup}, \mathsf{Gen}, \Pi_{\mathsf{Sign}}, \mathsf{KAg}, \mathsf{Vrfy})$*, formally defined as:*

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

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

$\Pi_{\mathsf{Sign}\langle sk_i, sk_{1-i}\rangle}(pk_0, pk_1, m)$**:** *is an interactive, PPT protocol that on input secret keys sk_i from party $\mathcal{P}_i$ with $i \in \{0,1\}$ and common values $m \in \{0,1\}^*$ and pk_0, pk_1, outputs a signature σ.*

$\mathsf{KAg}(pk_0, pk_1)$**:** *is a DPT algorithm that on input two public keys pk_0, pk_1, outputs an aggregated public key apk.*

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

The *completeness* property of SIG_2 guarantees that if the protocol Π_{Sign} is executed correctly between the two parties, the resulting signature is a valid signature under the aggregated public key.

Definition 8 (Completeness). *A two-party signature with aggregatable public keys* SIG_2 *satisfies completeness, if for all key pairs* $(sk, pk) \leftarrow \mathsf{Gen}(1^n)$ *and messages* $m \in \{0,1\}^*$*, the protocol* $\Pi_{\mathsf{Sign}\langle sk_i, sk_{1-i}\rangle}(pk_0, pk_1, m)$ *outputs a signature* σ *to both parties* $\mathcal{P}_0, \mathcal{P}_1$ *such that* $\mathsf{Vrfy}_{apk}(m;\sigma) = 1$ *where* $apk := \mathsf{KAg}(pk_0, pk_1)$.

A two-party signature scheme with aggregatable public keys should satisfy *unforgeability.* At a high level, this property guarantees that if one of the two parties is malicious, this party is not able to produce a valid signature under the aggregated public key without cooperation of the other party. We formalize the property through an experiment $\mathsf{SigForge}^b_{\mathcal{A},\mathsf{SIG}_2}$, where $b \in \{0,1\}$ defines which of the two parties is corrupt. This experiment is initialized by a security parameter n and run between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$, which proceeds as follows. The challenger first generates the public parameters pp by running the setup procedure $\mathsf{Setup}(1^n)$ as well as a signing key pair (sk_{1-b}, pk_{1-b}) by executing $\mathsf{Gen}(1^n)$, thereby simulating the honest party $\mathcal{P}_{1-b}$. Thereafter, $\mathcal{C}$ forwards pp_{C} and pk_{1-b} to the adversary $\mathcal{A}$ who generates its own key pair (sk_b, pk_b), thereby emulating the malicious party P_b, and submits (sk_b, pk_b) to $\mathcal{C}$. The adversary $\mathcal{A}$ additionally obtains access to an *interactive* and stateful signing oracle $\mathcal{O}^b_{\Pi_{\mathrm{S}}}$, which simulates the honest party $\mathcal{P}_{1-b}$ during the execution of $\Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-b},\cdot\rangle}$. Furthermore, every queried message m is stored in a query list $\mathcal{Q}$.

Eventually, $\mathcal{A}$ outputs a forgery in form of a $\mathsf{SIG}_2^{\mathsf{ID}}$ signature σ^* and a message m^*. $\mathcal{A}$ wins the experiment if σ^* is a valid signature for m^* under the aggregated public key $apk := \mathsf{KAg}(pk_0, pk_1)$ and m^* was never queried before, i.e., $m^* \notin \mathcal{Q}$. Below, we give a formal definition of the unforgeability game.

Definition 9 (2-EUF–CMA Security). *A two-party, public key aggregatable signature scheme* SIG_2 *is unforgeable if for every PPT adversary* $\mathcal{A}$*, there exists a negligible function* ν *such that: for* $b \in \{0,1\}$, $\Pr[\mathsf{SigForge}^b_{\mathcal{A},\mathsf{SIG}_2}(n) = 1] \leq \nu(n)$, *where the experiment* $\mathsf{SigForge}^b_{\mathcal{A},\mathsf{SIG}_2}(n)$ *is defined as follows:*

$\mathsf{SigForge}^b_{\mathcal{A},\mathsf{SIG}_2}(n)$	$\mathcal{O}^b_{\Pi_{\mathrm{S}}}(m)$
$1: \mathcal{Q} := \emptyset, pp \leftarrow \mathsf{Setup}(1^n)$	$1: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$2: (sk_{1-b}, pk_{1-b}) \leftarrow \mathsf{Gen}(pp)$	$2: \sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-b},\cdot\rangle}(pk_0, pk_1, m)$
$3: (sk_b, pk_b) \leftarrow \mathcal{A}(pp, pk_{1-b})$	
$4: (\sigma^*, m^*) \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_{\mathrm{S}}}(\cdot)}(pk_{1-b}, sk_b, pk_b)$	
$5: \mathbf{return}\ \left(m^* \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{\mathsf{KAg}(pk_0,pk_1)}(m^*;\sigma^*)\right)$	

Remark 3 (On security definition.). There are two different approaches for modeling signatures with aggregatable public keys in the literature, namely the plain

public-key model [3] (also known as key-verification model [12]) and the knowledge-of-secret-key (KOSK) model [7]. In the plain public-key setting the adversary chooses a key pair (sk_b, pk_b) and only declares the public key pk_b to the challenger in the security game. However, security proofs in this setting typically require rewinding techniques with the forking lemma. This is undesirable for the purpose of this paper, as we aim to construct adaptor signatures and its two-party variant generically as building blocks for further applications such as payment channels [2]. Payment channels are proven secure in the UC framework that does not allow the use of rewinding techniques in order to ensure concurrency. Thus, the plain public-key model does not seem suitable for our purpose. In the KOSK setting, however, the adversary outputs its (possibly maliciously chosen) key pair (sk_b, pk_b) to the challenger. In practice this means that the parties need to exchange zero-knowledge proofs of knowledge of their secret key[3]. Similar to previous works [7, 28], we do not require the forking lemma or rewinding in the KOSK setting and hence follow this approach.

4.2 Generic Transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to $\mathsf{SIG}_2^{\mathsf{ID}}$

We now give a generic transformation from $\mathsf{SIG}^{\mathsf{ID}}$ schemes to two-party signature schemes with aggregatable public keys.

At a high level, our transformation turns the signing procedure into an interactive protocol which is executed between the two parties $\mathcal{P}_0, \mathcal{P}_1$. The main idea is to let both parties engage in a randomness exchange protocol in order to generate a joint public randomness which can then be used for the signing procedure. In a bit more detail, to create a joint signature, each party $\mathcal{P}_i$ for $i \in \{0, 1\}$ can individually create a partial signature with respect to the *joint randomness* by using the secret key sk_i and exchange her partial signature with $\mathcal{P}_{1-i}$. The joint randomness ensures that both partial signatures can be combined to one jointly computed signature.

In the following, we describe the randomness exchange protocol that is executed during the signing procedure in more detail, as our transformation heavily relies on it. The protocol, denoted by $\mathit{\Pi}_{\mathsf{Rand\text{-}Exc}}$, makes use of two cryptographic building blocks, namely an extractable commitment scheme $\mathsf{C} = (\mathsf{Gen}, \mathsf{Com}, \mathsf{Dec}, \mathsf{Extract})$ and a NIZK proof system $\mathsf{NIZK} = (\mathsf{Setup_R}, \mathsf{Prove}, \mathsf{Verify})$. Consequently, the common input to both parties $\mathcal{P}_0$ and $\mathcal{P}_1$ are the public parameters pp_{C} of the commitment scheme, while each party P_i takes as secret input her secret key sk_i. In the following, we give description of the $\mathit{\Pi}_{\mathsf{Rand\text{-}Exc}\langle sk_0, sk_1\rangle}(pp_{\mathsf{C}}, \mathsf{crs})$ protocol and present it in a concise way in Fig. 5.

1. Party $\mathcal{P}_0$ generates her public randomness R_0 using algorithm P_1 from the underlying ID scheme alongside a NIZK proof $\pi_0 \leftarrow \mathsf{NIZK.Prove}(\mathsf{crs}, R_0, sk_0)$ that this computation was executed correctly with the corresponding secret value sk_0. $\mathcal{P}_0$ executes $(c, d) \leftarrow \mathsf{C.Com}(pp, (R_0, \pi_0))$ to commit to R_0 and π_0 and sends the commitment c to $\mathcal{P}_1$.

[3] Using techniques from [20, 16] it is possible to obtain NIZKs which allow for witness extraction without rewinding.

$\mathcal{P}_0(pp_{\mathsf{C}}, \mathsf{crs}, sk_0)$		$\mathcal{P}_1(pp_{\mathsf{C}}, \mathsf{crs}, sk_1)$
$(R_0, St_0) \leftarrow \mathsf{P}_1(sk_0)$		
$\pi_0 \leftarrow \mathsf{NIZK.Prove}(\mathsf{crs}, R_0, sk_0)$		
$(c, d) \leftarrow \mathsf{C.Com}(pp_{\mathsf{C}}, (R_0, \pi_0))$	$\xrightarrow{c}$	$(R_1, St_1) \leftarrow \mathsf{P}_1(sk_1)$
	$\xleftarrow{R_1, \pi_1}$	$\pi_1 \leftarrow \mathsf{NIZK.Prove}(\mathsf{crs}, R_1, sk_1)$
	$\xrightarrow{d}$	$R'_0 \leftarrow \mathsf{C.Dec}(pp_{\mathsf{C}}, c, d)$
If $\mathsf{NIZK.Verify}(\mathsf{crs}, R_1, \pi_1) = 0$, then abort		If $\mathsf{NIZK.Verify}(\mathsf{crs}, R'_0, \pi_0) = 0$, then abort
R_0, St_0, R_1		R_1, St_1, R_0

Fig. 5: $\mathit{\Pi}_{\mathsf{Rand\text{-}Exc}}$ Protocol

$\mathsf{Setup}(1^n)$

1 : $pp_{\mathsf{C}} \leftarrow \mathsf{C.Gen}(1^n)$
2 : $\mathsf{crs} \leftarrow \mathsf{NIZK.Setup_R}(1^n)$
3 : **return** $pp := (1^n, pp_{\mathsf{C}}, \mathsf{crs})$

$\mathsf{Gen}(pp)$

1 : Parse $pp = (1^n, pp_{\mathsf{C}}, \mathsf{crs})$
2 : $(sk, pk') \leftarrow \mathsf{IGen}(n)$
3 : $pk := (pp, pk')$
4 : **return** (sk, pk)

$\mathsf{KAg}(pk_0, pk_1)$

1 : $apk := f_{\mathsf{com\text{-}pk}}(pk_0, pk_1)$
2 : **return** apk

$\mathit{\Pi}_{\mathsf{Sign}\langle sk_i, sk_{1-i}\rangle}(pk_i, pk_{1-i}, m)$

1 : Parse $pk_i = ((1^n, pp_{\mathsf{C}}, \mathsf{crs}), pk'_i)$
2 : $(R_i, St_i, R_{1-i}) \leftarrow \mathit{\Pi}_{\mathsf{Rand\text{-}Exc}\langle sk_i, sk_{1-i}\rangle}(pp_{\mathsf{C}}, \mathsf{crs})$
3 : $R_{\mathsf{sign}} := f_{\mathsf{com\text{-}rand}}(R_0, R_1)$
4 : $h := \mathcal{H}(R_{\mathsf{sign}}, m)$
5 : $s_i \leftarrow \mathsf{P}_2(sk_i, R_i, h, St_i)$
6 : $s_{1-i} \leftarrow \mathit{\Pi}_{\mathsf{Exchange}}\langle s_i, s_{1-i}\rangle$
7 : $(h, s) := f_{\mathsf{com\text{-}sig}}(h, (s_0, s_1))$
8 : **return** (h, s)

$\mathsf{Vrfy}_{apk}(m; (h, s))$

1 : $R_{\mathsf{sign}} := \mathsf{V}_0(apk, h, s)$
2 : **return** $h := \mathcal{H}(R_{\mathsf{sign}}, m)$

Fig. 6: $\mathsf{SIG}_2^{\mathsf{ID}}$: SIG_2 scheme from identification scheme.

2. Upon receiving the commitment c from $\mathcal{P}_0$, party $\mathcal{P}_1$ generates her public randomness R_1 using algorithm P_1. She also computes a NIZK proof as $\pi_1 \leftarrow \mathsf{NIZK.Prove}(\mathsf{crs}, R_1, sk_1)$, which proves correct computation of R_1, and sends R_1 and π_1 to $\mathcal{P}_0$.
3. Upon receiving R_1 and π_1 from $\mathcal{P}_1$, $\mathcal{P}_0$ sends the opening d to her commitment c to $\mathcal{P}_1$.
4. $\mathcal{P}_1$ opens the commitment in this round. At this stage, both parties check that the received zero-knowledge proofs are valid. If the proofs are valid, each party $\mathcal{P}_i$ for $i \in \{0, 1\}$ outputs R_i, St_i, R_{1-i}.

Our transformation can be found in Fig. 6. Note that we use a deterministic function $f_{\mathsf{com\text{-}rand}}(\cdot, \cdot)$ in step 3 in the signing protocol which combines the two public random values R_0 and R_1. In step 6 of the same protocol, we assume that the partial signatures are exchanged between the parties via the protocol $\mathit{\Pi}_{\mathsf{Exchange}}$ upon which the parties can combine them using a deterministic function $f_{\mathsf{com\text{-}sig}}(\cdot, \cdot)$ in step 7. Further, a combined signature can be verified under a combined public key of the two parties. In more detail, to verify a combined signature $(h, s) := f_{\mathsf{com\text{-}sig}}(h, (s_0, s_1))$,

in step 7, there must exist an additional deterministic function $f_{\mathsf{com\text{-}pk}}(\cdot,\cdot)$ (in step 1 of the KAg algorithm) such that:

$$\Pr\left[\mathsf{Vrfy}_{apk}(m;(h,s))=1 \middle| \begin{array}{l}(pk_0,sk_0)\leftarrow\mathsf{IGen}(n),(pk_1,sk_1)\leftarrow\mathsf{IGen}(n)\\ (h,s)\leftarrow\Pi_{\mathsf{Sign}\langle sk_0,sk_1\rangle}(pk_0,pk_1,m)\\ apk:=f_{\mathsf{com\text{-}pk}}(pk_0,pk_1)\end{array}\right]=1. \quad (4)$$

We also require that given a full signature and a secret key sk_i with $i\in\{0,1\}$, it is possible to extract a valid partial signature under the the public key pk_{1-i} of the other party. In particular, there exists a function $f_{\mathsf{dec\text{-}sig}}(\cdot,\cdot,\cdot)$ such that:

$$\Pr\left[\mathsf{Vrfy}_{pk_{1-i}}(m;(h,s_{1-i}))=1 \middle| \begin{array}{l}(pk_0,sk_0)\leftarrow\mathsf{IGen}(n),(pk_1,sk_1)\leftarrow\mathsf{IGen}(n)\\ (h,s)\leftarrow\Pi_{\mathsf{Sign}\langle sk_0,sk_1\rangle}(pk_0,pk_1,m)\\ (h,s_{1-i}):=f_{\mathsf{dec\text{-}sig}}(sk_i,pk_i,(h,s))\end{array}\right]=1. \quad (5)$$

Note that equations 4 and 5 implicitly define $f_{\mathsf{com\text{-}sig}}$ through the execution of Π_{Sign} in the conditional probabilities.

The instantiations of these functions for Schnorr, Katz-Wang signatures and Guillou-Quisquater signatures can be found in the full version of this paper [14].

We note the similarity between this transformation with that in Fig. 3. In particular, both of them compute the public randomness R_{sign} by shifting the original random values. Note also that running the algorithm V_0 on the inputs (pk_i,h,s_i) would return $R_i, \forall i\in\{0,1\}$.

Below, we show that the transformation in Fig. 6 provides a secure two-party signature with aggregatable public keys. To this end, we show that $\mathsf{SIG}_2^{\mathsf{ID}}$ satisfies SIG_2 *completeness* and *unforgeability* from Def. 8 and Def. 9, respectively.

Theorem 3. *Assume that $\mathsf{SIG}^{\mathsf{ID}}$ is a signature scheme based on the transformation from an identification scheme as per Fig. 2. Further, assume that the functions $f_{\mathsf{com\text{-}sig}}$, $f_{\mathsf{com\text{-}pk}}$ and $f_{\mathsf{dec\text{-}sig}}$ satisfy the relations, Equations (4) and (5) respectively. Then the resulting $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme from the transformation in Fig. 6 is a secure two-party signature scheme with aggregatable public keys in the random oracle model.*

Lemma 5. *Under the assumptions of Thm. 3, $\mathsf{SIG}_2^{\mathsf{ID}}$ satisfies Def. 8.*

Proof. The proof follows directly from Eq. 4 and the construction of KAg algorithm in Fig. 6.

Lemma 6. *Under the assumptions of Thm. 3, $\mathsf{SIG}_2^{\mathsf{ID}}$ satisfies Def. 9.*

Proof. We prove this lemma by exhibiting a simulator $\mathcal{S}$ that breaks the unforgeability of the $\mathsf{SIG}^{\mathsf{ID}}$ scheme if it has access to an adversary that can break the unforgeability of the $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme. More precisely, we show a series of games, starting with the $\mathsf{SigForge}^b_{\mathcal{A},\mathsf{SIG}_2}$ experiment, such that each game is computationally indistinguishable from the previous one. The last game is modified in such a way that the simulator

can use the adversary's forgery to create its own forgery for the unforgeability game against the $\mathsf{SIG}^{\mathsf{ID}}$ scheme.

To construct this simulator, we note that the $\Pi_{\mathsf{Rand\text{-}Exc}}$ protocol in Fig. 6 must satisfy two properties (similar to [27]). First, the commitment scheme must be extractable for the simulator, and second, the NIZK proof used must be simulatable. The reasons for these two properties become evident in the proof.

We prove Lemma 6 by separately considering the cases of the adversary corrupting party $\mathcal{P}_0$ or party $\mathcal{P}_1$, respectively.

Adversary corrupts $\mathcal{P}_0$. In the following we give the security proof in case the adversary corrupts party $\mathcal{P}_0$.

Game G_0: This is the regular $\mathsf{SigForge}^0_{\mathcal{A},\mathsf{SIG}_2}(n)$ experiment, in which the adversary plays the role of party $\mathcal{P}_0$. In the beginning of the game, the simulator generates the public parameters as $pp \leftarrow \mathsf{Setup}(1^n)$. Note that the Setup procedure, apart from computing $\mathsf{crs} \leftarrow \mathsf{NIZK.Setup_R}(1^n)$, includes the execution of $\mathsf{C.Gen}$ through which the simulator learns the trapdoor tr for the commitment scheme C. Further, $\mathcal{S}$ generates a fresh signing key pair $(sk_1, pk_1) \leftarrow \mathsf{Gen}(1^n)$, sends pp and pk_1 to $\mathcal{A}$ and receives the adversary's key pair (pk_0, sk_0). The simulator simulates the experiment honestly. In particular, it simulates the interactive signing oracle $\mathcal{O}^0_{\Pi_\mathsf{S}}$ honestly by playing the role of party $\mathcal{P}_1$.

Game G_1: This game proceeds exactly like the previous game, with a modification in the simulation of the signing oracle. Upon $\mathcal{A}$ initiating the signing protocol by calling the interactive signing oracle, $\mathcal{S}$ receives the commitment c to its public randomness R_0 from $\mathcal{A}$. The simulator, using the trapdoor tr, then extracts a randomness $R_0' \leftarrow \mathsf{C.Extract}(pp, tr, c)$ and computes the joint randomness as $R_{\mathsf{sign}} \leftarrow f_{\mathsf{com\text{-}rand}}(R_0', R_1)$. $\mathcal{S}$ honestly computes the zero-knowledge proof to its own randomness R_1 and sends it to $\mathcal{A}$. Upon receiving the opening d to c from the adversary, $\mathcal{S}$ checks if $R_0' = \mathsf{C.Dec}(pp, c, d)$. If this does not hold, $\mathcal{S}$ aborts, otherwise $\mathcal{S}$ continues to simulate the rest of the experiment honestly.

Claim. Let Bad_1 be the event that G_1 aborts in the signing oracle. Then, we have $\Pr[\mathsf{Bad}_1] \leq \nu_1(n)$, where ν_1 is a negligible function in n.

Proof: Note that game G_1 aborts only if the extracted value R_0' from commitment c is not equal to the actual committed value R_0 in c, i.e., if $\mathsf{C.Extract}(pp, tr, c) \neq \mathsf{C.Dec}(pp, c, d)$. By the extractability property of C this happens only with negligible probability. In other words, it holds that $\Pr[\mathsf{Bad}_1] \leq \nu_1(n)$, where ν_1 is a negligible function in n. ■

Game G_2: This game proceeds as game G_1, with a modification to the signing oracle. Upon input message m, instead of generating its signature (h, s_0) with respect to the joint public randomness R_{sign}, the simulator generates it only with respect to its own randomness R_0. Further, the simulator programs the random oracle in the following way: as in the previous game, it computes the joint randomness R_{sign} and then programs the random oracle in a way such that on input (R_{sign}, m) the random oracle returns h.

It is easy to see that this game is indistinguishable from $\boldsymbol{G_1}$ if the adversary has not queried the random oracle on input (R_{sign}, m) before the signing query. If, however, the adversary has issued this random oracle query before the signing query (i.e., $\mathcal{H}(R_{\mathsf{sign}}, m) \neq \bot$)), then the simulation aborts.

Claim. Let Bad_2 be the event that $\boldsymbol{G_2}$ aborts in the signing oracle. Then, we have $\Pr[\mathsf{Bad}_2] \leq \nu_2(n)$, where ν_2 is a negligible function in n.

Proof: We first recall that the output of P_1 (i.e., R_{pre}) is uniformly random from a super-polynomial set of size q in the security parameter. From this it follows that R_{sign} is distributed uniformly at random in the same set. Furthermore, $\mathcal{A}$ being a PPT algorithm, can only make polynomially many queries to $\mathcal{H}$ and $\mathcal{O}_{\mathrm{pS}}$ oracles. Denoting ℓ as the total number of queries to $\mathcal{H}$ and $\mathcal{O}_{\mathrm{S}}$, we have: $\Pr[\mathsf{Bad}_2] = \Pr[\mathcal{H}(R_{\mathsf{sign}}, m) \neq \bot] \leq \frac{\ell}{q} \leq \nu_2(n)$. This follows from the fact that ℓ is polynomial in the security parameter. ∎

Game $\boldsymbol{G_3}$: In this game, the only modification as compared to the previous game is that during the Setup procedure, the simulator executes the algorithm $(\widetilde{\mathsf{crs}}, \tau) \leftarrow \mathsf{NIZK.Setup}'_{\mathsf{R}}(1^n)$ instead of $\mathsf{crs} \leftarrow \mathsf{Setup}_{\mathsf{R}}(1^n)$, which allows the simulator to learn the trapdoor τ. Since the two distributions $\{\mathsf{crs} : \mathsf{crs} \leftarrow \mathsf{Setup}_{\mathsf{R}}(1^n)\}$ and $\{\widetilde{\mathsf{crs}} : (\widetilde{\mathsf{crs}}, \tau) \leftarrow \mathsf{Setup}'_{\mathsf{R}}(1^n)\}$ are indistinguishable to $\mathcal{A}$ except with negligible probability, we have that $\Pr[\boldsymbol{G_2} = 1] \leq \Pr[\boldsymbol{G_3} = 1] + \nu_3(n)$ where ν_3 is a negligible function in n.

Game $\boldsymbol{G_4}$: This game proceeds exactly like the previous game except that the simulator does not choose its own key pair, but rather uses its signing oracle from the $\mathsf{EUF{-}CMA}$ game to simulate the adversary's interactive signing oracle $\mathcal{O}^0_{\Pi_{\mathrm{S}}}$. More concretely, upon the adversary calling $\mathcal{O}^0_{\Pi_{\mathrm{S}}}$ on message m, the simulator calls its own signing oracle which provides a signature (h, s_1) for m under secret key sk_1. Note that the simulator does not know sk_1 or the secret randomness r_1 used in s_1. Therefore, the simulator has to additionally simulate the NIZK proof that proves knowledge of r_1 in s_1. More concretely, the simulator executes $\pi_{\mathsf{S}} \leftarrow \mathsf{S}(\widetilde{\mathsf{crs}}, \tau, R_1)$, where R_1 is the public randomness used in s_1. Due to the fact that the distributions $\{\pi : \pi \leftarrow \mathsf{Prove}(\widetilde{\mathsf{crs}}, R_1, r_1)\}$ and $\{\pi_{\mathsf{S}} : \pi_{\mathsf{S}} \leftarrow \mathsf{S}(\widetilde{\mathsf{crs}}, \tau, R_1)\}$ are indistinguishable to $\mathcal{A}$ except with negligible probability, it holds that $\Pr[\boldsymbol{G_3} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \nu_4(n)$ where ν_4 is a negligible function in n.

It remains to show that the simulator can use a valid forgery output by $\mathcal{A}$ to break unforgeability of the $\mathsf{SIG}^{\mathsf{ID}}$ scheme.

Claim. A valid forgery $(m^*, (h^*, s^*))$ output by $\mathcal{A}$ in game $\mathsf{SigForge}_{\mathcal{A},\mathsf{SIG}_2^{\mathsf{ID}}}$ can be transformed into a valid forgery $(m^*, (h^*, s_1^*))$ in game $\mathsf{SigForge}_{\mathcal{S},\mathsf{SIG}^{\mathsf{ID}}}$.

Proof: When $\mathcal{A}$ outputs a valid forgery $(m^*, (h^*, s^*))$, $\mathcal{S}$ extracts the partial signature (h^*, s_1^*) by executing $f_{\mathsf{dec\text{-}sig}}(sk_0, pk_0, (h^*, s^*))$ (from Eq. 5). Note that the simulator knows the adversary's key pair (sk_0, pk_0). The simulator then submits $(m^*, (h^*, s_1^*))$ as its own forgery to the $\mathsf{EUF{-}CMA}$ challenger. By definition, $\mathcal{A}$ wins this game if it has not queried a signature on m^* before. Thus, $\mathcal{S}$ has also not queried the $\mathsf{EUF{-}CMA}$ signing oracle on m^* before. Further, Eq. (5) implies that $(m^*, (h^*, s_1^*))$ is a valid forgery under the public key pk_1. ∎

From games $\boldsymbol{G_0} - \boldsymbol{G_4}$, we have that $\Pr[\boldsymbol{G_0} = 1] \leq \Pr[\boldsymbol{G_4} = 1] + \nu(n)$, where $\nu(n) = \nu_1(n) + \nu_2(n) + \nu_3(n) + \nu_4(n)$ is a negligible function in n. Thus, we have $\Pr[\mathsf{SigForge}_{\mathcal{A},\mathsf{SIG}_2^{\mathsf{ID}}}(n) = 1] \leq \Pr[\mathsf{SigForge}_{\mathcal{S},\mathsf{SIG}^{\mathsf{ID}}}(n) = 1] + \nu(n)$.

Adversary corrupts $\mathcal{P}_1$. In case the adversary corrupts $\mathcal{P}_1$, the simulator has to simulate $\mathcal{P}_0$. The proof for this case follows exactly the same steps as above with the exception that game $\boldsymbol{G_1}$ is not required. This is due to the reason that the simulator now plays the role of the committing party in the randomness exchange and hence does not have to extract $\mathcal{A}$'s randomness from the commitment c.

5 Two-party Aggregatable Adaptor Signatures

We are now ready to formally introduce the notion of two-party adaptor signatures with aggregatable public keys which we denote by aSIG_2. Our definition can be seen as a combination of the definition of adaptor signatures from Sec. 3 and the definition of two-party signatures with aggregatable public keys from Sec. 4. Unlike the single party adaptor signatures, in aSIG_2 both parties have the role of the signer and generate pre-signatures cooperatively. Furthermore, both parties can adapt the pre-signature given a witness value y. We note that both the pre-signature and the full-signature are valid under the aggregated public keys of the two parties. We formally define an aSIG_2 scheme w.r.t. a SIG_2 scheme (which is in turn defined w.r.t. a SIG scheme) and a hard relation R.

Afterwards, we show how to instantiate our new definition. Concretely, we present a generic transformation that turns a $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme with certain homomorphic properties into a two-party adaptor signatures scheme. As a $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme is constructed w.r.t. a $\mathsf{SIG}^{\mathsf{ID}}$ scheme (cf. Sec. 4), the construction presented in this section can implicitly transform digital signatures based on ID schemes to two-party adaptor signatures.

The definition of a two-party adaptor signature scheme aSIG_2 is similar to the definition of a standard adaptor signature scheme as for Def. 1. The main difference lies in the pre-signature generation. Namely, the algorithm pSign is replaced by a *protocol* Π_{pSign} which is executed between two parties.

Definition 10 (Two-Party Adaptor Signature Scheme with Aggregatable Public Keys). *A two-party adaptor signature scheme with aggregatable public keys is defined w.r.t. a hard relation* R *and a two-party signature scheme with aggregatable public keys* $\mathsf{SIG}_2 = (\mathsf{Setup}, \mathsf{Gen}, \Pi_{\mathsf{Sign}}, \mathsf{KAg}, \mathsf{Vrfy})$. *It is run between parties* $\mathcal{P}_0, \mathcal{P}_1$ *and consists of a tuple* $\mathsf{aSIG}_2 = (\Pi_{\mathsf{pSign}}, \mathsf{Adapt}, \mathsf{pVrfy}, \mathsf{Ext})$ *of efficient protocols and algorithms which are defined as follows:*

$\Pi_{\mathsf{pSign}\langle sk_i, sk_{1-i}\rangle}(pk_0, pk_1, m, Y)$**:** *is an interactive protocol that on input secret keys* sk_i *from party* $\mathcal{P}_i$ *with* $i \in \{0,1\}$ *and common values public keys* pk_i, *message* $m \in \{0,1\}^*$ *and statement* $Y \in L_{\mathsf{R}}$, *outputs a pre-signature* $\widetilde{\sigma}$.

$\mathsf{pVrfy}_{apk}(m, Y; \widetilde{\sigma})$**:** *is a DPT algorithm that on input an aggregated public key apk, a message* $m \in \{0,1\}^*$, *a statement* $Y \in L_{\mathsf{R}}$ *and a pre-signature* $\widetilde{\sigma}$, *outputs a bit* b.

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

$\mathsf{Ext}_{apk}(\sigma, \widetilde{\sigma}, Y)$**:** *is a DPT algorithm that on input an aggregated public key apk, a signature* σ*, pre-signature* $\widetilde{\sigma}$ *and statement* $Y \in L_{\mathsf{R}}$*, outputs a witness* y *such that* $(Y, y) \in \mathsf{R}$*, or* $\perp$*.*

We note that in aSIG_2, the pVrfy algorithm enables public verifiability of the pre-signatures, e.g., aSIG_2 can be used in a three-party protocol where the third party needs to verify the validity of the generated pre-signatrue.

In the following, we formally define properties that a two-party adaptor signature scheme with aggregatable public keys aSIG_2 has to satisfy. These properties are similar to the ones for single party adaptor signature schemes. We start by defining two-party pre-signature correctness which, similarly to Def. 2 states that an honestly generated pre-signature and signature are valid, and it is possible to extract a valid witness from them.

Definition 11 (Two-Party Pre-Signature Correctness). *A two-party adaptor signature with aggregatable public keys* aSIG_2 *satisfies two-party pre-signature correctness, if for all* $n \in \mathbb{N}$*, messages* $m \in \{0,1\}^*$*, it holds that:*

$$\Pr\left[\begin{array}{c|l} \mathsf{pVrfy}_{apk}(m, Y; \widetilde{\sigma}) = 1 & pp \leftarrow \mathsf{Setup}(1^n), (sk_0, pk_0) \leftarrow \mathsf{Gen}(pp) \\ \wedge & (sk_1, pk_1) \leftarrow \mathsf{Gen}(pp), (Y, y) \leftarrow \mathsf{GenR}(1^n) \\ \mathsf{Vrfy}_{apk}(m; \sigma) = 1 & \widetilde{\sigma} \leftarrow \Pi_{\mathsf{pSign}\langle sk_0, sk_1\rangle}(pk_0, pk_1, m, Y) \\ \wedge & apk := \mathsf{KAg}(pk_0, pk_1) \\ (Y, y') \in \mathsf{R} & \sigma := \mathsf{Adapt}_{apk}(\widetilde{\sigma}, y), y' := \mathsf{Ext}_{apk}(\sigma, \widetilde{\sigma}, Y) \end{array}\right] = 1.$$

The unforgeability security definition is similar to Def. 9, except the adversary interacts with two oracles $\mathcal{O}^b_{\Pi_{\mathrm{S}}}, \mathcal{O}^b_{\Pi_{\mathrm{pS}}}$ in order to generate signatures and pre-signatures, as in Def. 3. More precisely, in the $\mathsf{aSigForge}^b_{\mathcal{A},\mathsf{aSIG}_2}(n)$ experiment defined below, $\mathcal{A}$ obtains access to *interactive*, stateful signing and pre-signing oracles $\mathcal{O}^b_{\Pi_{\mathrm{S}}}$ and $\mathcal{O}^b_{\Pi_{\mathrm{pS}}}$ respectively. Oracles $\mathcal{O}^b_{\Pi_{\mathrm{S}}}$ and $\mathcal{O}^b_{\Pi_{\mathrm{pS}}}$ simulate the honest party $\mathcal{P}_{1-b}$ during an execution of the protocols $\Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-\mathsf{b}}, \cdot\rangle}$ and $\Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-\mathsf{b}}, \cdot\rangle}$ respectively. Similar to Def. 9, both the protocols $\Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-\mathsf{b}}, \cdot\rangle}, \Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-\mathsf{b}}, \cdot\rangle}$ employed by the respective oracles $\mathcal{O}^b_{\Pi_{\mathrm{S}}}, \mathcal{O}^b_{\Pi_{\mathrm{pS}}}$ gets an oracle access to $\mathcal{A}$ as well.

Definition 12 (2-aEUF–CMA Security). *A two-party adaptor signature with aggregatable public keys* aSIG_2 *is unforgeable if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* ν *such that:* $\Pr[\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_2}(n) = 1] \leq \nu(n)$*, where the experiment* $\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_2}(n)$ *is defined as follows:*

The definition of two-party pre-signature adaptability follows Def. 4 closely. The only difference is that in this setting the pre-signature must be valid under the aggregated public keys.

Definition 13 (Two-Party Pre-Signature Adaptability). *A two-party adaptor signature scheme with aggregatable public keys* aSIG_2 *satisfies two-party pre-signature adaptability, if for all* $n \in \mathbb{N}$*, messages* $m \in \{0,1\}^*$*, statement and*

$\mathsf{aSigForge}^b_{\mathcal{A},\mathsf{aSIG}_2}(n)$	$\mathcal{O}^b_{\Pi_\mathrm{S}}(m)$
$1: \mathcal{Q} := \emptyset, pp \leftarrow \mathsf{Setup}(1^n)$	$1: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$2: (sk_{1-b}, pk_{1-b}) \leftarrow \mathsf{Gen}(pp)$	$2: \sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-b},\cdot\rangle}(pk_0, pk_1, m)$
$3: (sk_b, pk_b) \leftarrow \mathcal{A}(pp, pk_{1-b})$	
$4: m^* \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}, \mathcal{O}^b_{\Pi_\mathrm{pS}}}(pk_{1-b}, sk_b, pk_b)$	$\mathcal{O}^b_{\Pi_\mathrm{pS}}(m, Y)$
$5: (Y, y) \leftarrow \mathsf{GenR}(1^n)$	$1: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$6: \widetilde{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-b},\cdot\rangle}(m^*, Y)$	$2: \widetilde{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-b},\cdot\rangle}(pk_0, pk_1, m, Y)$
$7: \sigma^* \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}, \mathcal{O}^b_{\Pi_\mathrm{pS}}}(\widetilde{\sigma}, Y)$	
$8: \textbf{return } \Big(m^* \notin \mathcal{Q} \wedge \mathsf{Vrfy}_{\mathsf{KAg}(pk_0, pk_1)}(m^*; \sigma^*)\Big)$	

witness pairs $(Y, y) \in \mathsf{R}$*, public keys* pk_0 *and* pk_1*, and pre-signatures* $\widetilde{\sigma} \in \{0,1\}^*$ *satisfying* $\mathsf{pVrfy}_{apk}(m, Y; \widetilde{\sigma}) = 1$ *where* $apk := \mathsf{KAg}(pk_0, pk_1)$*, we have* $\Pr[\mathsf{Vrfy}_{apk}(m; \mathsf{Adapt}_{apk}(\widetilde{\sigma}, y)) = 1] = 1$.

Finally, we define two-party witness extractability.

Definition 14 (Two-Party Witness Extractability). *A two-party public key aggregatable adaptor signature scheme* aSIG_2 *is* witness extractable *if for every PPT adversary* $\mathcal{A}$*, there exists a negligible function* ν *such that the following holds:* $\Pr[\mathsf{aWitExt}_{\mathcal{A},\mathsf{aSIG}_2}(n) = 1] \leq \nu(n)$*, where the experiment* $\mathsf{aWitExt}_{\mathcal{A},\mathsf{aSIG}_2}$ *is defined as follows:*

$\mathsf{aWitExt}^b_{\mathcal{A},\mathsf{aSIG}_2}(n)$	$\mathcal{O}^b_{\Pi_\mathrm{S}}(m)$
$1: \mathcal{Q} := \emptyset, pp \leftarrow \mathsf{Setup}(1^n)$	$1: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$2: (sk_{1-b}, pk_{1-b}) \leftarrow \mathsf{Gen}(pp)$	$2: \sigma \leftarrow \Pi^{\mathcal{A}}_{\mathsf{Sign}\langle sk_{1-b},\cdot\rangle}(pk_0, pk_1, m)$
$3: (sk_b, pk_b) \leftarrow \mathcal{A}(pp, pk_{1-b})$	
$4: (m^*, Y^*) \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}, \mathcal{O}^b_{\Pi_\mathrm{pS}}}(pk_{1-b}, sk_b, pk_b)$	$\mathcal{O}^b_{\Pi_\mathrm{pS}}(m, Y)$
$5: \widetilde{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-b},\cdot\rangle}(m^*, Y^*)$	$1: \mathcal{Q} := \mathcal{Q} \cup \{m\}$
$6: \sigma^* \leftarrow \mathcal{A}^{\mathcal{O}^b_{\Pi_\mathrm{S}}, \mathcal{O}^b_{\Pi_\mathrm{pS}}}(\widetilde{\sigma})$	$2: \widetilde{\sigma} \leftarrow \Pi^{\mathcal{A}}_{\mathsf{pSign}\langle sk_{1-b},\cdot\rangle}(pk_0, pk_1, m, Y)$
$7: apk := \mathsf{KAg}(pk_0, pk_1), y' := \mathsf{Ext}_{apk}(\sigma^*, \widetilde{\sigma}, Y^*)$	
$8: \textbf{return } (m^* \notin \mathcal{Q} \wedge (Y^*, y') \notin \mathsf{R} \wedge \mathsf{Vrfy}_{apk}(m^*; \sigma^*))$	

Note that the only difference between this experiment and the $\mathsf{aSigForge}_{\mathcal{A},\mathsf{aSIG}_2}$ experiment is that here the adversary is allowed to choose the statement/witness pair (Y^*, y^*) and that the winning condition additionally requires that for the extracted witness $y' \leftarrow \mathsf{Ext}_{apk}(\sigma^*, \widetilde{\sigma}, Y^*)$ it holds that $(Y^*, y') \notin \mathsf{R}$.

A two-party adaptor signature scheme with aggregatable public keys aSIG_2 is called *secure* if it satisfies 2-aEUF–CMA security, *two-party pre-signature adaptability* and *two-party witness extractability* properties.

5.1 Generic transformation from $\mathsf{SIG}_2^{\mathsf{ID}}$ to $\mathsf{aSIG}_2^{\mathsf{ID,R}}$

We now present our generic transformation to achieve two-party adaptor signature schemes with aggregatable public keys from identification schemes. In its essence, this transformation is a combination of the transformations presented in Figs. 3 and 6. More precisely, similar to the transformation from $\mathsf{SIG}^{\mathsf{ID}}$ to $\mathsf{aSIG}^{\mathsf{ID,R}}$ presented in Fig. 3, we assume the existence of functions f_{shift}, f_{adapt} and f_{ext} with respect to the relation R. We then make use of the $\Pi_{\mathsf{Rand\text{-}Exc}}$ protocol from the transformation in Fig. 6 to let parties agree on the randomness that is going to be used during the pre-signing process. However, unlike the transformation in Fig. 6, the resulting randomness is shifted by a statement Y for relation R using the function f_{shift}. The transformation can be found in Fig. 7.

$\Pi_{\mathsf{pSign}\langle sk_0, sk_1\rangle}(pk_0, pk_1, m, Y)$

1 : Parse $pk_i = ((1^n, pp_{\mathsf{C}}, \mathsf{crs}), pk_i'), i \in \{0,1\}$
2 : $(R_i, St_i, R_{1-i}) \leftarrow \Pi_{\mathsf{Rand\text{-}Exc}\langle sk_i, sk_{1-i}\rangle}(pp_{\mathsf{C}}, \mathsf{crs})$
3 : $R_{\mathsf{pre}} := f_{\mathsf{com\text{-}rand}}(R_0, R_1)$
4 : $R_{\mathsf{sign}} := f_{\mathsf{shift}}(R_{\mathsf{pre}}, Y), h := \mathcal{H}(R_{\mathsf{sign}}, m)$
5 : $\tilde{s}_i \leftarrow \mathsf{P}_2(sk_i, R_i, h, St_i)$
6 : $\tilde{s}_{1-i} \leftarrow \Pi_{\mathsf{Exchange}}\langle \tilde{s}_i, \tilde{s}_{1-i}\rangle$
7 : $(h, \tilde{s}) := f_{\mathsf{com\text{-}sig}}(h, (\tilde{s}_i, \tilde{s}_{1-i}))$
8 : **return** $(h, \tilde{s})$

$\mathsf{pVrfy}_{apk}(m, Y; (h, \tilde{s}))$

1 : $\widehat{R}_{\mathsf{pre}} := \mathsf{V}_0(apk, h, \tilde{s})$
2 : **return** $h = \mathcal{H}(f_{\mathsf{shift}}(\widehat{R}_{\mathsf{pre}}, Y), m)$

$\mathsf{Adapt}_{pk}((h, \tilde{s}), y)$

1 : **return** $(h, f_{\mathsf{adapt}}(\tilde{s}, y))$

$\mathsf{Ext}_{pk}((h, s), (h, \tilde{s}), Y)$

1 : **return** $f_{\mathsf{ext}}(s, \tilde{s})$

Fig. 7: A two-party adaptor signature scheme with aggregatable public keys $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ defined with respect to a $\mathsf{SIG}_2^{\mathsf{ID}}$ scheme and a hard relation R.

Theorem 4. *Assume that* $\mathsf{SIG}^{\mathsf{ID}}$ *is an* $\mathsf{SUF\text{-}CMA}$*-secure signature scheme transformed using Fig. 2. Let* $f_{\mathsf{shift}}, f_{\mathsf{adapt}}$ *and* f_{ext} *be functions satisfying the relations from Equations (1) and (2), and* R *be a hard relation. Further, assume that* $f_{\mathsf{com\text{-}sig}}$, $f_{\mathsf{com\text{-}pk}}$ *and* $f_{\mathsf{dec\text{-}sig}}$ *satisfy the relation from Equations (4) and (5). Then the resulting* $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ *scheme from the transformation in Fig. 7 is a secure two-party adaptor signature scheme with aggregatable public keys in the random oracle model.*

In order to prove Thm. 4, we must show that $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ satisfies the *pre-signature correctness*, 2-aEUF–CMA *security*, *pre-signature adaptability* and *witness extractability* properties as described in Defs. 11 to 14 respectively. We provide the full proofs of the following lemmas in the full version of this paper [14] and only mention the intuition behind the proofs here. As mentioned in the introduction of this work, despite the fact that $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ is constructed from $\mathsf{SIG}_2^{\mathsf{ID}}$, we require only $\mathsf{SIG}^{\mathsf{ID}}$ to be SUF–CMA-secure in order to prove 2-aEUF–CMA security for $\mathsf{aSIG}_2^{\mathsf{ID,R}}$.

Lemma 7 (Two-Party Pre-Signature Correctness). *Under the assumptions of Thm. 4,* $\mathsf{aSIG}_2^{\mathsf{ID,R}}$ *satisfies Def. 11.*

The proof of Lemma 7 follows directly from Equations (1) to (3) and the correctness of SIG_2 from Lemma 5.

Lemma 8 (2-$\mathsf{aEUF–CMA}$-Security). *Under the assumptions of Thm. 4,* $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ *satisfies Def. 12.*

Proof Sketch: In a nutshell the proof of this lemma is a combination of the proofs of Lemmas 2 and 6, i.e., the proof is done by a reduction to the hardness of the relation R and the $\mathsf{SUF–CMA}$ of the underlying signature scheme. During the signing process, the challenger queries its $\mathsf{SUF–CMA}$ signing oracle and receives a signature σ. As in the proof of Lemma 6, the challenger programs the random oracle such that σ appears like a signature generated with the combined randomness of the challenger and the adversary. Simulating the pre-signing process is similar with the exception that before programming the random oracle, the randomness must be shifted using the function f_{shift}. Finally, the challenger and the adversary generate a pre-signature $\widetilde{\sigma}^* = (h, \tilde{s})$ on the challenge message m^* and the adversary outputs the forgery $\sigma^* = (h, s)$. If $f_{\mathsf{ext}}(s, \tilde{s})$ returns the y generated by the challenger, as in the proof of Lemma 2, the hardness of the relation R can be broken. Otherwise, using $f_{\mathsf{dec\text{-}sig}}$, it is possible to use the forgery provided by the adversary to extract a forgery for the $\mathsf{SUF–CMA}$ game.

Lemma 9 (Two-Party Pre-Signature Adaptability). *Under the assumptions of Thm. 4,* $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ *satisfies Def. 13.*

Proof Sketch: The proof of Lemma 9 is analogous to the proof of Lemma 3.

Lemma 10 (Two-party Witness Extractability). *Under the assumptions of Thm. 4,* $\mathsf{aSIG}_2^{\mathsf{ID},\mathsf{R}}$ *satisfies Def. 14.*

Proof Sketch: The proof of Lemma 10 is very similar to the proof of Lemma 8 except that the adversary chooses Y now and thus, no reduction to the hardness of the relation R is needed.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) Emmy Noether Program *FA 1320/1-1*, by the *DFG CRC 1119 CROSSING* (project S7), by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902) and by *NSCX project* (with the project number CS1920241NSCX008801) on *Design and Development of Blockchain based Technologies* in the *Department of Computer Science and Engineering, IIT Madras.*

References

[1] M. Abdalla et al. “Tighter Reductions for Forward-Secure Signature Schemes”. In: *PKC 2013*. 2013.

[2] L. Aumayr et al. *Generalized Bitcoin-Compatible Channels*. Cryptology ePrint Archive, Report 2020/476. https://eprint.iacr.org/2020/476.pdf. 2020.

[3] M. Bellare and G. Neven. "Multi-signatures in the plain public-Key model and a general forking lemma". In: *ACM CCS 2006*. 2006.

[4] M. Bellare and S. Shoup. "Two-Tier Signatures, Strongly Unforgeable Signatures, and Fiat-Shamir Without Random Oracles". In: *PKC 2007*. 2007.

[5] *Bitcoin Scripts*. https://en.bitcoin.it/wiki/Script#Crypto.

[6] *Bitcoin Wiki: Payment Channels*. https://en.bitcoin.it/wiki/Payment_channels.

[7] A. Boldyreva. "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme". In: *PKC 2003*. 2003.

[8] D. Boneh et al. "Compact Multi-signatures for Smaller Blockchains". In: *ASIACRYPT 2018, Part II*. 2018.

[9] D. Boneh et al. "Short Signatures from the Weil Pairing". In: *ASIACRYPT 2001*. 2001.

[10] C. Decker and R. Wattenhofer. "A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels". In: *Stabilization, Safety, and Security of Distributed Systems 2015*. 2015.

[11] A. Deshpande and M. Herlihy. "Privacy-Preserving Cross-Chain Atomic Swaps". In: *FC 2020*. 2020.

[12] M. Drijvers et al. "On the Security of Two-Round Multi-Signatures". In: *2019 IEEE Symposium on Security and Privacy*. 2019.

[13] L. Eckey et al. *Splitting Payments Locally While Routing Interdimensionally*. Cryptology ePrint Archive, Report 2020/555. https://eprint.iacr.org/2020/555. 2020.

[14] A. Erwig et al. *Two-Party Adaptor Signatures From Identification Schemes*. Cryptology ePrint Archive, Report 2021/150. https://eprint.iacr.org/2021/150. 2021.

[15] M. F. Esgin et al. "Post-Quantum Adaptor Signatures and Payment Channel Networks". In: *ESORICS 2020*. 2020.

[16] M. Fischlin. "Communication-Efficient Non-interactive Proofs of Knowledge with Online Extractors". In: *CRYPTO 2005*. 2005.

[17] L. Fournier. *One-Time Verifiably Encrypted Signatures A.K.A. Adaptor Signatures*. https://github.com/LLFourn/one-time-VES/blob/master/main.pdf. 2019.

[18] R. Gennaro et al. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *ACNS 16*. 2016.

[19] S. Goldwasser and R. Ostrovsky. "Invariant signatures and non-interactive zero-knowledge proofs are equivalent". In: *Annual International Cryptology Conference*. Springer. 1992.

[20] J. Groth et al. "Perfect Non-interactive Zero Knowledge for NP". In: *EUROCRYPT 2006*. 2006.

[21] J. Gugger. *Bitcoin–Monero Cross-chain Atomic Swap*. Cryptology ePrint Archive, Report 2020/1126. https://eprint.iacr.org/2020/1126. 2020.

[22] L. C. Guillou and J.-J. Quisquater. “A “Paradoxical” Indentity-Based Signature Scheme Resulting from Zero-Knowledge”. In: *CRYPTO’88*. 1990.

[23] T. Hardjono and Y. Zheng. “A practical digital multisignature scheme based on discrete logarithms (extended abstract)”. In: *Advances in Cryptology — AUSCRYPT ’92*. 1993.

[24] J. Katz and N. Wang. “Efficiency Improvements for Signature Schemes with Tight Security Reductions”. In: *ACM CCS 2003*. 2003.

[25] E. Kiltz et al. “Optimal Security Proofs for Signatures from Identification Schemes”. In: *CRYPTO 2016, Part II*. 2016.

[26] D.-P. Le et al. *DDH-based Multisignatures with Public Key Aggregation*. Cryptology ePrint Archive, Report 2019/771. https://eprint.iacr.org/2019/771. 2019.

[27] Y. Lindell. “Fast Secure Two-Party ECDSA Signing”. In: *CRYPTO 2017, Part II*. 2017.

[28] S. Lu et al. “Sequential Aggregate Signatures and Multisignatures Without Random Oracles”. In: *EUROCRYPT 2006*. 2006.

[29] A. Lysyanskaya. “Unique signatures and verifiable random functions from the DH-DDH separation”. In: *Annual International Cryptology Conference*. Springer. 2002.

[30] G. Malavolta et al. “Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability”. In: *NDSS 2019*. 2019.

[31] G. Maxwell et al. “Simple Schnorr multi-signatures with applications to Bitcoin”. In: *Designs, Codes and Cryptography 2019* (2019).

[32] S. Micali et al. “Verifiable Random Functions”. In: *40th FOCS*. 1999.

[33] A. Miller et al. “Sprites and State Channels: Payment Networks that Go Faster Than Lightning”. In: *FC 2019*. 2019.

[34] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. http://bitcoin.org/bitcoin.pdf. 2009.

[35] K. Ohta and T. Okamoto. “A digital multisignature scheme based on the Fiat-Shamir scheme”. In: *Advances in Cryptology — ASIACRYPT ’91*. 1993.

[36] T. Okamoto. “A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems”. In: *ACM Trans. Comput. Syst.* 4 (1988).

[37] A. Poelstra. *Scriptless scripts*. https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2017-03-mit-bitcoin-expo/slides.pdf. 2017.

[38] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-chain Instant Payments*. https://lightning.network/lightning-network-paper.pdf. 2016.

[39] R. L. Rivest et al. “How to Leak a Secret”. In: *ASIACRYPT 2001*. 2001.

[40] N. van Saberhagen. *CryptoNote v 2.0*. https://bytecoin.org/old/whitepaper.pdf.

[41] C.-P. Schnorr. “Efficient Signature Generation by Smart Cards”. In: *Journal of Cryptology* 3 (1991).

[42] S.-T. Shen et al. “Unique signature with short output from cdh assumption”. In: *International Conference on Provable Security*. Springer. 2015.

[43] E. Tairi et al. *A^2L: Anonymous Atomic Locks for Scalability in Payment Channel Hubs*. Cryptology ePrint Archive, Report 2019/589. https://eprint.iacr.org/2019/589. 2019.

[44] E. Tairi et al. *Post-Quantum Adaptor Signature for Privacy-Preserving Off-Chain Payments*. To appear at FC 2021. https://eprint.iacr.org/2020/1345.

D. Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma

This chapter corresponds to our published article in the 12th Innovations in Theoretical Computer Science (ITCS), 2021 [62] (https://doi.org/10.4230/LIPIcs.ITCS.2021.72). Our full version can be found in [61].

Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma*

Stefan Dziembowski[1], Grzegorz Fabiański[1], Sebastian Faust[2], and Siavash Riahi[2]

[1] University of Warsaw
[2] TU Darmstadt

Abstract. Blockchain is a disruptive new technology introduced around a decade ago. It can be viewed as a method for recording timestamped transactions in a public database. Most of blockchain protocols do not scale well, i.e., they cannot process quickly large amounts of transactions. A natural idea to deal with this problem is to use the blockchain only as a timestamping service, i.e., to hash several transactions $tx_1, \ldots, tx_m$ into one short string, and just put this string on the blockchain, while at the same time posting the hashed transactions $tx_1, \ldots, tx_m$ to some public place on the Internet ("off-chain"). In this way the transactions tx_i remain timestamped, but the amount of data put on the blockchain is greatly reduced. This idea was introduced in 2017 under the name *Plasma* by Poon and Buterin. Shortly after this proposal, several variants of Plasma have been proposed. They are typically built on top of the Ethereum blockchain, as they strongly rely on so-called *smart contracts* (in order to resolve disputes between the users if some of them start cheating). Plasmas are an example of so-called *off-chain protocols*.
In this work we initiate the study of the inherent limitations of Plasma protocols. More concretely, we show that in every Plasma system the adversary can either (a) force the honest parties to communicate a lot with the blockchain, even though they did not intend to (this is traditionally called *mass exit*); or (b) an honest party that wants to leave the system needs to quickly communicate large amounts of data to the blockchain. What makes these attacks particularly hard to handle in real life is that these attacks do not have so-called *uniquely attributable faults*, i.e. the smart contract cannot determine which party is malicious, and hence cannot force it to pay the fees for the blockchain interaction. An important implication of our result is that the benefits of two of the most prominent Plasma types, called *Plasma Cash* and *Fungible Plasma*, cannot be achieved simultaneously.
Besides of the direct implications on real-life cryptocurrency research, we believe that this work may open up a new line of theoretical research, as, up to our knowledge, this is the first work that provides an impossibility result in the area of off-chain protocols.

1 Introduction

What does it mean to *timestamp* a digital document? Haber and Stornetta in their seminal paper [17] define timestamping as a method to certify when a given document was created. In many settings the timestamped document T remains secret after it was timestamped, until its creator decides to make it public. This is often because of efficiency reasons – for example in the scheme of [17] what is really timestamped is the cryptographic hash[3] $H(T)$ (not T), which leads to savings in

* This work was partly supported by the FY18-0023 PERUN from the Ethereum Foundation, by the TEAM/2016-1/4 grant from the Foundation for Polish Science, by the DFG CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research (BMBF) iBlockchain project (grant nr. 16KIS0902), and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

[3] In this paper we assume reader's familiarity with basic cryptographic notions such as hash functions, negligible functions, etc. For an introduction to this topic see, e.g., [20].

communication between T's creator and the timestamping service. Sometimes the secrecy of T is actually a desired feature. According to [29] several researchers in the past (including Galileo Galilei and Isaac Newton) have used ad-hoc methods to timestamp their research ideas before publishing them, in order to later claim priority.

Recently, in the context of blockchain, timestamping has been used in a slightly different way. Namely, in the paper that introduced this technology [25], the "timestamping" mechanism is such that T's creator does not only get a proof that T has been created at a given time, but also that it has been *made public* at this time. Let us call this kind of scheme *public timestamping*, and let us refer to the former type of timestamping as *secret* timestamping. The public timestamping feature of blockchain has been one of the main reasons why this technology attracted so much attention. In fact one of the first projects that use blockchain for purposes other than purely financial was *Namecoin* that used the timestamping to create a decentralized domain name system (see, e.g., [19] for more on this project).

In many cases timestamping is expensive, and its costs grow linearly with the length of the timestamped document. This is especially true for blockchain-based solutions, where all parties in the network need to reach *consensus* about what document was published and when. For example, Bitcoin (the blockchain system introduced in [25]) can process at most around 1MB of data per 10 minutes. In Ethereum, which is another very popular blockchain system [31] "timestamping" a word of 32 bytes cost currently around USD 0.80. A similar problem appears in several other blockchain protocols. We give a short introduction to blockchain in Sec. 1.1. For a moment let us just say that typical blockchains come with their own virtual currencies (also called *cryptocurrencies*). In the most standard case the "timestamped" messages define financial transfers between the network participants, and are hence called *transactions*. We will refer to timestamping a transaction as *posting it on the blockchain*.

To summarize, from the efficiency point of view the secret timestamping is better than the public one (as only hashes need to be timestamped). From the security perspective these two types of timestamping are incomparable as the fact that the timestamped document T needs to be published can be considered to be both an advantage and a disadvantage, depending on the particular application. For example, one could argue that considering timestamped hashes of academic papers as being sufficient evidence for claiming priority would slow down scientific progress, as it would disincentive making the papers public (until, say, the author writes down all followup papers that build upon the timestamped paper).

A natural question therefore is as follows. Suppose we have access to a timestamping service that is expensive to use (so we would rather timestamp only very short messages there). Can we use it to "emulate" public timestamping that would be cheap for long documents T? Of course this "emulation" cannot work in general, since some scenarios may simply require a proof that the whole T was available publicly at a given time. So the best we can hope for is to find *some* applications which permit such emulation. An idea for such emulation emerged recently within

the cryptocurrency community under the names *Plasma* or *commit chains*[4] [26, 21, 16, 28, 24]. Very informally speaking, Plasma allows to "compress" a number of blockchain transactions $tx_1, \ldots, tx_m$ into one very short string $h = H(T)$, where $T := (tx_1, \ldots, tx_m)$. The transactions can from from different parties called *users*. The only document that gets posted on the blockchain is h. The compression and posting h is done by one designated party called the *operator* (denoted Op). Besides of posting h, the operator publishes all the transactions T on some public network (say: on her web-page). Since this publication is *not* done on the blockchain we also say that it is performed *off-chain*.

Publishing T off-chain is important since only then each user can verify that her transaction was indeed included into the hashed value. Moreover, typical Plasma designs use as H the so-called *Merkle-tree hashing* with $tx_1, \ldots, tx_m$ being the labels of the leaves (see the full version of this paper [11] for more on this technique). Thanks to this, every user can prove that his transaction was included in $H(T)$ with a proof of length $O(\log n)$. If the operator does not include some tx_i in T then the user U that produced tx_i can always post tx_i directly to the blockchain. In this case using Plasma does not bring any benefits to U compared to just using the blockchain directly from the beginning. However, in reality it is expected that this is not going to happen often, especially since the operators are envisioned to be commercial entities that will charge some fee for their services.

What is much more problematic is the case when the malicious operator does *not* publish T off-chain. This situation is typically referred to as *data unavailability*. Note that data unavailability is subjective, i.e., the parties can have different views on whether it happened or not. This is because, unlike the situation on the blockchain, there may be no consensus on what was published off-chain. Moreover, there is no way to produce a proof that data unavailability happens: even if some parties complain on the blockchain that they did not receive T there is no way to determine (just by looking at the blockchain) whether they are right, or if they are just falsely accusing the operator. The situation when a disagreement between parties happens but there is no "blockchain-only" way to verify which party is corrupt, is commonly referred to as a *non-uniquely attributable fault*. Finally, some Plasma protocols require all the honest parties to immediately act on blockchain after a data unavailability attack happened. This is called "mass exit", although (for the reasons explained in Sec. 1.1 in this paper we call it "large forced on-chain action")

Plasma comes in many variants and has been discussed in countless articles (see Sec. 1.1). One of the most fundamental distinction is between the two types of Plasma systems: *Plasma Cash* and *Fungible Plasma*. As we explain in more detail in Sec. 1.1 they both serve for the "emulation" that we outlined above, but have different incomparable features. From one point of view Fungible Plasma is better than Plasma Cash since it is "fungible", which means that the money can be arbitrarily divided and merged. On the other hand: Fungible Plasma suffers from some problems that Plasma Cash does not have, namely the adversary can cause a "non-uniquely attributable large forced mass actions" (we explain these notions Sec. 1.1).

[4] In this paper we mostly use the name "Plasma" due to its brevity.

The cryptocurrency community has been unsuccessfully looking for a Plasma solution that would have the benefits of both Plasma Cash and Fungible Plasma. The main result of this paper is that such Plasma cannot be achieved simultaneously. In other words, we show inherent limitations of the compression technique, at least for compressing blockchain financial transactions. Our paper can also be viewed as initiating the theoretical analysis of lower bounds for the smart contract protocols. We write more about our contribution in Sec. 1.2. First, however, let us provide some more introduction to blockchain and to Plasmas.

1.1 Introduction to Plasma

Let us start with providing some more background on blockchain and smart contracts. Blockchain can be viewed as a public *ledger* containing timestamped transactions that have to satisfy some correctness constraints. Moreover, several blockchains permit to execute the so-called *smart contracts* [27] (or simply: "contracts"), which informally speaking, are "self-executable" agreements described in form of computer programs. Examples of such blockchain platforms include *Ethereum*, *Hyperledger Fabric*, or *Cardano*. Typically, it is assumed that contracts are deterministic and have a public state. Moreover, they can own some coins. Executing a contract is done by posting transactions on the blockchain and it costs fees that depend on the computational complexity of the given operation, and on the amount of data that needs to be transmitted to the contract.

Let us now explain the basic idea of Plasma, and introduce some standard terminology. Since it is an informal presentation, we mix the definition of the protocol with its construction. In the formal sections of the paper these two parts are separated (the definition appears in Sec. 2, and the constructions in the full version of this paper [11]). As highlighted above Plasma address the scalability problem of blockchain by keeping the massive bulk of transactions outside of the blockchain ("off-chain"). The parties that are involved in the protocol rely on a smart contract that is deployed on the ledger of the underlying cryptocurrency, but they try to minimize interacting with it. Typically, this interaction happens only when the parties join and leave the protocol, or when they disagree. Since all parties know that in case of disagreement, disputes can always be settled on the ledger, there is no incentive for the users to disagree, and honest behavior is enforced.

In the optimistic case, when the parties involved in the protocol play honestly, and the off-chain transactions never hit the ledger, these protocols significantly reduce transaction fees and allow for instantaneous executions. Off-chain protocols also resemble an idea explored in cryptography around two decades ago under the name "optimistic protocols" [7, 1]. In this model the parties are given access to a trusted server that is "expensive to use", and hence they do not want to contact it, unless it is absolutely necessary

Plasma's operator *Op* provides a "simulated ledger", in which other parties can deposit their coins, and then perform operations between each other. The key requirement is that its users do not need to trust the operator, and in particular if they discover that she is cheating, then they can safely withdraw their funds. The

latter is called an *exit* from the simulated ledger, and requires communication with the underlying ledger.

Plasma protocols come in different variants (see Sec. 1.1), however, they are all based on a single framework proposed in [26]. The parties that execute Plasma are: the *users* $U_1, \ldots, U_n$, and the *operator* Op. Moreover, the parties have access to a contract on the blockchain. In our formal modeling this contract will be represented as a trusted interactive machine Γ with public state, owning some amount of coins. Each user U_i has some number of coins initially deposited in his Plasma account which is maintained by Γ. This number is called a *balance* and is denoted with $b_i \in \mathbb{Z}_{\geq 0}$. Users' balances are changing dynamically during the execution of the protocol. The total number of coins owned by the contract Γ is equal to the sum of all balances of its users. A vector $\overrightarrow{b} := (b_1, \ldots, b_n)$ is called a *Plasma chain*. When referring to the underlying blockchain (i.e. the one on which Γ is deployed) we use the term *main chain*. Note that the operator Op has no account and only facilitates transfers of the users. In some variants of Plasma (see Sec. 1.1) the operator blocks some amount of coins (called operator's *collateral*) that can be used to compensate the users their losses in case she misbehaves.

Let us briefly describe the different operations that parties of the Plasma protocol can execution during the lifetime of the system. We divide time into *epochs* (e.g. 1 epoch takes 1 hour). In the ith epoch the operator sends some information C_i to Γ. We can think of C_i as "compressed" information about the vector $(b_1, \ldots, b_n)$ containing the users' balances. By "compressed" we mean that $|C_i|$ is much shorter than the description of $(b_1, \ldots, b_n)$, and usually its length is constant in every epoch. We will refer to C_i as a "commitment" to $(b_1, \ldots, b_n)$. The length of C_i is called the *commit size*.

In each epoch every user U_i can request to *exit*, by which we mean that all b_i coins from her Plasma account get converted to the "real" coins on the main chain, and she is no longer a part of this Plasma chain (which, in our formal modeling will be indicated by setting $b_i := \bot$). Plasma's security properties guarantee that every user can exit with all the coins that she currently has in the given Plasma chain. It is often required that exiting can be done cheaply, and in particular that the total length of the messages sent by the exiting user to Γ is small. The amount of data that a user needs to send to Γ in order to exit the Plasma chain is called the *exit size*.

Finally, any two users of the same Plasma chain can make *transfers* between each other. Suppose U_k wants to transfer v coins from her account to U_j. This transfer operation involves only communicating with U_j, and with the operator Op, while no interaction with the contract Γ is needed. Under normal circumstances (i.e. when the operator is honest) the next Plasma block that is committed to the main chain will simply have v coins deduced from U_k's account and v coins added to U_j's account.[5]

[5] To keep things simple, in this paper we do not discuss things like "transfer receipts", i.e., confirmations for the sender that the coins have been transferred.

Challenges in designing Plasma systems. The main challenge when designing a Plasma system is to guarantee that every user can exit with her money. This is usually achieved as follows: each C_i is a commitment to $(b_1, \ldots, b_n)$, computed using a Merkle tree. An honest operator Op is obliged to obey the following rule:

Explaining commitments — each time Op sends C_i to Γ, she sends the corresponding $\overrightarrow{b} := (b_1, \ldots, b_n)$ to all the users.

Technically, sending $\overrightarrow{b}$ to the users can be realized, e.g., by publishing it on the operator's web-page (i.e.: "off-chain"). Every user U_j can now check if she has the correct amount on her account and if C_i was computed correctly. Moreover, thanks to the properties of Merkle trees, U_j has a short proof of size $O(\log(n))$ that b_j has been "committed" into C_i.

The above description assume that the operator is honest. If she is corrupt things get more complicated. Note that C_i sent by the operator to Γ is publicly known (due to the properties of the underlying blockchain). Hence, we can assume that all the users agree on whether C_i was published and what is its exact value. The situation is different when it comes to the vector $\overrightarrow{b}$ that should be published off-chain. In particular, if $\overrightarrow{b}$ has *not* been published, then the users have no way to prove this to Γ. This is because whether some data has been published off-chain or not does not have a digital evidence that can be interpreted by Γ. This leads us to the following attack that can be carried out by a malicious operator, and is intensively studied by the cryptocurrency community (see, e.g., [5]).

Data unavailability attack — in this attack the corrupt operator publishes C_i but does not publish $\overrightarrow{b}$.

Note that this attack has no extra cost for the operator because from the point of view of the smart contract Γ, the operator behaves honestly, and hence the users cannot complain to Γ and request, e.g., that Op sends $\overrightarrow{b}$ to Γ. Furthermore determining if this attack happened is "subjective", i.e., every user U_j has to detect it herself. Moreover, in case of data unavailability it is impossible for Γ to determine whether U_j or Op is dishonest, since a sheer declaration of U_j that Op did not send him the data obviously cannot serve as a proof that it indeed happened. This leads to the following definition.

Non-uniquely attributable faults — this refers to the situation when the contract has to intervene in the execution of the protocol (because the protocol is under attack), but it unable to determine which party misbehaved (see, e.g., [2]).

Non-uniquely attributable faults appear typically in situations when a party claims that it has not received a message from another party. In contrast if a contract *is* able to determine which party is corrupt then we have a *uniquely*-attributable fault. A typical example of such a fault is when a party signs two contradictory messages. Unfortunately, non-uniquely attributable faults are hard to handle in real life, since it is not clear which party should pay the fees for executing the smart contract, or which party should be punished for misbehaving. In particular, what is unavoidable

in such a case is that a malicious party P can force another participant P' to lose money on fees (potentially also loosing money herself). This phenomenon is known as *griefing* [2].

When a user realizes that the operator is dishonest, then she often needs to start quickly interacting with Γ in order to protect her coins. This action has to be done quickly, and has to be performed by each honest user. This leads to the following definition.

Forced on-chain action of size α — this term refers to the situation that honest parties who did not intend to perform an exit are forced by the adversary to quickly interact with Γ, and the total length of the messages sent by them to Γ is α. Informally, when α is large (e.g. $\alpha = \Omega(n)$) we say this is a *mass* forced on-chain action.

Note that this definition talks about all honest "parties", and hence it includes also the case when the operator is honest, but it is forced to act because of the behavior of the corrupt users. Typically $\alpha = \Omega(n)$ and by "quickly" we mean "1 epoch". In most Plasma proposals [23, 26, 3] "interacting with the smart contract" means simply exiting the Plasma chain with all the coins. Hence, a more common term for this situation is "mass exit". Since in our work we are dealing with the lower bounds, we need to be ready to cover also other, non-standard, ways of protecting honest users' coins. For example, it could be the case that a user U_j does *not* exit immediately, but, instead, keeps her coins in a special account "within Γ" and withdraws them much later. Of course, this requires interacting with Γ immediately, but, technically speaking does not require "exiting". To capture such situations, we use the term "forced on-chain *action*", instead of "mass *exit*".

After a party announces an exit, we need to ensure that she is exiting with the right amount of coins. The main problem comes from the fact that we cannot require that users exit from the last C_i by sending the explanation for her balance b_i to Γ. This is because it could be the case that a given user does not know the explanation $\overrightarrow{b}$ of C_i (due to the data unavailability attack). For a description of how this can be done in practice see [26, 3], or the full version of this paper [11].

Mass exits (or large forced on-chain actions) caused by data unavailability are considered a major problem for Plasma constructions. They are mentioned multiple times in the original Plasma paper [26] (together with some ad-hoc mechanism for mitigating them). They are also routinely discussed on "Ethereum's Research Forum"[6], with even conferences organized on this topic[7]. One of the main reasons why the mass exits are so problematic is that they may results in blockchain congestion (i.e., situations when too many users want to send transactions to the underlying blockchain). Moreover, the adversary can choose to attack Plasma precisely in the moments when the blockchain is already close to being congested (see, e.g., [30] for a description of real-life incident of the Ethereum blockchain congestion). She can also attack different Plasma chains (established over the same main chain) so

[6] Available at: `ethresear.ch`.

[7] See: `ethresear.ch/t/data-unavailability-unconference-devcon4`.

their users simultaneously send large amounts of data to the blockchain. In order to be prepared for such events in real-life Plasma proposals it is sometimes suggested that the time T for reacting to data unavailability should be very large (e.g. $T = 2$ weeks). This, unfortunately, has an important downside, namely that also an honest coin withdrawal requires time T.

Consider a non-fungible Plasma that supports coin identifiers from some set $\mathcal{C}$. In a non-fungible Plasma the Plasma chain is of a different form than before: instead of a vector of balances $\overrightarrow{b}$, it is a function $f : \mathcal{C} \to \{U_1, \ldots, U_n, \bot\}$ that assigns to every coin $c \in \mathcal{C}$ its current owner $f(c)$ (or $\bot$ if the coin has been withdrawn). Similar to before the commitment to the value of f will be done using Merkle trees. Whenever a coin is withdrawn its identifier c is sent to the smart contract Γ, and hence it becomes public. This is important for the mechanism that prevents parties from stealing coins. To this end, each user U monitors Γ, and sends a complaint whenever some (corrupt) user U' tries to withdraw one of U's coins. For the contract Γ to decide if c belongs to U or U' can require some additional interaction, but the system is designed in such a way that the honest user is guaranteed that finally she will win such dispute. Hence, every malicious attempt to withdraw someone else's coin will be stopped.

The main difference between Plasma Cash and Fungible Plasma is that in Plasma Cash every user has to "protect" only her own coins. Thanks to this, even in case of the data unavailability attack, each honest user U does *not* need to immediately take any action. Instead, she can just monitor Γ, and has to act only if someone tries to withdraw one of U's coins. Of course, the corrupt user can still force all the honest ones to quickly act on the blockchain. However, this requires much more effort from them than in Fungible Plasma, namely: they need to withdraw many coins of the honest users at once, hence forcing the honest users to react. This is "fairer" both honest and malicious users have to make similar effort. Most importantly, however, this attack has *uniquely-attributable faults*.

This advantage of Plasma Cash comes at a price, namely the "exit size" is not constant anymore, as it depends on the number of coins that a user has (since each coin has to be withdrawn "independently"). The Ethereum research community has been making some efforts to deal with this problem. One promising approach is to "compress" the information about withdrawn coins. For example one could assume that the identifiers in $\mathcal{C}$ are natural numbers. Then a user U who owns coins from some interval $[a, \ldots, b]$ (with $a, b \in \mathbb{N}$) could simply withdraw them by posting a message "User U withdraws all coins from the interval $[a, \ldots, b]$ (instead of withdrawing each $i \in [a, \ldots, b]$ independently). This, of course, works only if the coins that users own can be divided into such intervals. Some authors (in particular V. Buterin) have been suggesting "defragmentation" techniques for achieving such a distribution of coins. This is based on the assumption that the parties periodically *cooperate* to "clean up" the system. Hence, it does not work in a fully malicious settings[8] (if the goal of the adversary is to prevent the cleaning procedure).

[8] See `ethresear.ch/t/plasma-cash-defragmentation` and subsequent posts by Buterin on the Ethereum Research Forum.

The landscape of Plasmas. Soon after the original groundbreaking work on Plasma [26], some concrete variants have been proposed. Some of them we already described in Sec. 1.1. Since this paper focuses on the *impossibility* results, we do not provide a complete overview of the many different variants that exist and what features they achieve. Plasma projects that are frequently mentioned in the media are Loom, Bankex, NOCUST and OmiseGO [28, 24]. This area is mostly developed by a very vibrant on-line community that typically communicates results in form of so-called "white-papers", blog articles, or post on discussion forums (such as the "Ethereum Research Forum", see footnote 6 on page 7). See also the diagram called "Plasma World Map" illustrating the different flavors of Plasma in the full version of this paper [11]. A notable exception are NOCUST and NOCUST-ZKP described in [21]. This work, up to our knowledge, is the first academic paper on this topic. It provides a formal protocol description (with several interesting innovations such as " Merkle interval trees") and a security argument. Moreover, the authors of [21] describe a version of NOCUST that puts a collateral on the operator (this is done in order to achieve instant transaction confirmation). The authors of [21] (see also [16]) introduce the term "commit chains". Yet, unfortunately, they do not define a full formal security model that we could re-use in our work.

Let us also mention some of the so-called "distributed exchanges" that look very similar to Plasma. One example is StarkDEX (informally described in [15]), which is also based on the idea of a central operator batching transactions using Merkle trees, and a procedure for the users to "escape" from the system if something goes wrong. This protocol uses non-interactive zero-knowledge protocols to ensure correctness of the operator's actions (similar approach has been informally sketched in the original Plasma paper [26], and has been also used in NOCUST-ZKP [21]). While zero knowledge can be used to demonstrate that some data was computed correctly, it *cannot* be used to prove that the off-chain data *was published at all*. Consequently, the authors of this system also encountered the challenge of handling the data unavailability attack. Currently, in StarkDEX this problem is solved by introducing an external committee that certifies if data is available.[9] StarkDEX plans to eventually replace the committee-based solution with an approach that is only based on trusting the underlying blockchain. Our result however shows that in general this will be impossible, as long as fungibility and short exits are required (unless the operator puts a huge collateral).

1.2 Our contribution and organization of the paper

We initiate the study of lower bounds (or: "impossibility results") in the area of off-chain protocols. Our results can also be viewed as a part of a general research program of "bringing order to Plasma". We believe that the scientific cryptographic community can provide significant help in the efforts to systematize this area, and to determine the formal security guarantees of the protocols (in a way that is similar to the work on "Bitcoin backbone" [14], or more recently on "Mimblewimble" [13],

[9] See their FAQs at `https://www.starkdex.io`.

state channels [9], or the Lightning Network [22]). Investigating the limits of what Plasma can achieve is part of this process. We focus on proving lower bounds that concern the necessity of mass forced on-chain actions, especially caused by the attack that have no uniquely attributable faults (as a result of data unavailability). This is motivated by the fact that such attacks are particularly important for the off-chain protocols: since the main goal of such protocols is to move the transactions *off*-chain, the necessity of quickly acting *on*-chain can be viewed as a big disadvantage.

We start with a formal definition of Plasma (this is done in Sec. 2). Since in this work we are interested only in the impossibility results, our definition is very restrictive for practical systems. By "restrictive" we mean that we make several assumptions about how the protocol operates. For example we have very strict synchronicity rules, and in particular we only allow the users to start the Plasma operations in certain moments (see "payment orders" and "exit orders" phases in Sec. 2.1). Obviously, such restrictions make our lower bounds only stronger, since they also apply to a more realistic model (without such restrictions). We believe that fully formalizing real-life Plasma (e.g. in the style of [10, 12, 22]) is an important future research project, but it is beyond the scope of this work.

Our main result is presented in Thm. 1, stated formally in Sec. 3. It states that in certain cases the adversary can force mass on-chain actions of the honest users of *any* Plasma system. One subtle point that we want to emphasize is that whenever we talk about "forcing actions" on the honest users, we mean a situation when the users that did *not* want to exit (in a given epoch) are forced to act on-chain. This is important, as otherwise our theorem would hold trivially (one can always imagine a scenario when lots of users decide to exit Plasma because of some other, external, reasons). The notion of "wanting to exit" is formalized by an *environment machine* $\mathcal{Z}$ (borrowed from the *Universal Composability* framework [8]) that "orders" the parties to behave in certain way.

More formally, Thm. 1 states that in Plasma either there exists an attack that provokes a mass action, or there is an attack that requires a party that exits to post long messages on the blockchain (i.e. this Plasma has large exits). Moreover, both attacks have *no* uniquely attributable faults. Note that, strictly speaking, this theorem also covers Plasma systems where the commit size is large (even $\Omega(n)$)[10], but in this case it holds trivially since the honest operator needs to send the large commitments to Γ even if everybody is honest (hence: there is an "unprovoked" mass action in every epoch).

The most interesting practical implication of this theorem is that it confirms the need for "two different" Plasma flavors, as long as the operator is not required to put aside a collateral of size comparable to the total amount of coins in the system[11]. One way to look at it is: either we want to have a Plasma system that does not have large exits, in which case we need to have (non-uniquely attributable) mass

[10] In practice, Plasma systems with unbounded commit size are not interesting since they do not bring any advantages to the users. Moreover, they can be trivially constructed just by putting every transaction on the main chain.

[11] This is clearly impractical for most of the applications. Actually most of Plasma constructions assume no such collateral at all.

actions (this is Fungible Plasma/Plasma MVP); or alternatively we insist on having Plasma without such mass actions, but then we have to live with large exits (as in Plasma Cash). Our theorem implies that there is no Plasma that would combine the benefits of both Fungible Plasma and Plasma Cash, and hence can serve as a justification why both approaches are complementary. Before our result one could hope that the opposite is true and that, e.g., the only reason why Plasma Cash is popular is its relative simplicity (compared to Fungible Plasma). Besides of this reason, and the general scientific interest, we believe that our lower bound has some other important practical applications. In particular, lower bounds often serve as a guideline for constructing new systems or tweaking the definitions. We hope it will contribute to consolidate the countless research efforts in constructing new Plasma systems[12] and simplify identifying proposals that are not sound (e.g., because they claim to achieve the best of both worlds).

Let us also stress that our Thm. 1 does *not* rely on any assumptions of complexity-theoretic type and does *not* use a concept of "black-box separations" [18]. This means that the lower bound that we prove cannot be circumvented by introducing any kind of strong cryptographic assumptions. Hence, of course, it also holds for Plasmas that use non-interactive zero-knowledge (like NOCUST-ZKP [21] or StarkDEX [15]). Moreover, we manage to generalize our lower bound. Thm. 1 even holds for Plasma systems where the operator deposits a certain amount of coins for compensating parties for malicious behavior (e.g., it could be used when a malicious operator does not explain commitments).

For completeness, we also describe (in the full version of this paper [11]) "positive" results, i.e., two protocols that satisfy our security definition (Plasma Cash and Fungible Plasma). We stress that we do not consider it to be a part of our main contribution, and we do not claim novelty with these constructions, as they strongly rely on ideas published earlier (in particular [26, 4, 23, 3, 21, 15]).

Notation. For a formal definition of an *interactive (Turing) machine* and a *protocol*, see, e.g., [8]. In our modeling the communication between the parties is synchronous and happens in *rounds* (see Sect. 2). During the execution of the protocol a party P may send messages to a party P'. A *transcript* of the messages sent from P to P' is a sequence $\{(m_i, t_i)\}_{i=1}^{\ell}$, where each m_i was sent by P to P' in the t_i-th round. A transcript of messages sent from some set of parties to a different set of parties is a sequence $\{(W_i, W_i', m_i, t_i)\}_{i=1}^{\ell}$, where each m_i was sent by W_i to W_i' in the t_i-th round. By the *length of a transcript* we mean its bit-length (in some fixed encoding). We sometimes refer to it also as *communication size* (between the parties).

2 Plasma Payment Systems

A *Plasma payment system* (or "Plasma" for short) is a protocol Π consisting of a randomized non-interactive machine Ψ representing the *setup* of the system; deter-

[12] see `https://ethresear.ch/c/plasma/`.

ministic[13] interactive poly-time machines $U_1, \ldots, U_n, Op$ representing the *users* and the *operator* of the Plasma system (respectively); and a deterministic interactive poly-time machine Γ, which represents the *Plasma contract*. We use the notation $\mathcal{U} = \{U_1, \ldots, U_n\}$ to refer to the set of users of the system. The contract machine Γ has no secret state, and moreover its entire execution history is known to all the parties. We can think of it as a Turing machine that keeps the entire log of its execution history, and moreover all the other parties in the system have a (read-only) access to this log. The Plasma system comes with a parameter $\gamma \in \mathbb{R}_{\geq 0}$ called *operator's collateral fraction*. Informally, this parameter describes the amount of coins that are held by the operator as a "collateral" (as a fraction of user's coins). These coins can be used to cover users' losses if the operator misbehaves. This is formally captured in Sect. 2.2 (see "limited responsibility of the operator"). If $\gamma = 0$ then we say that the operator is *not collateralized*. We introduce the notion of collateral in order to make our results stronger and to cover also cases of real-life systems that have such a collateral (e.g., NOCUST, see Sect. 1.1).

The protocol is attacked by a randomized poly-time adversary $\mathcal{A}$. We assume that $\mathcal{A}$ can corrupt any number of users and the operator except the contract Γ (hence Γ can be seen as a trusted third party). Once $\mathcal{A}$ corrupts a party P, she learns all its secrets, and takes full control over it (i.e. she can send messages on behalf of P). A party that has not been corrupted is called *honest*. An execution of a Plasma payment system Π is parametrized by the *security parameter* 1^λ.

To model the fact that users perform actions, we use the concept of an environment $\mathcal{Z}$ (which is also a poly-time machine) that is responsible for "orchestrating" the execution of the protocol. The environment can send and receive messages from all the parties (it also has full access to the state of Γ). It also knows which parties are corrupt and which are honest. For an adversary $\mathcal{A}$ and an environment $\mathcal{Z}$, a pair $(\mathcal{A}, \mathcal{Z})$ will be called an *attack* (on a given Plasma system Π). The contract machine Γ can output special messages $(\mathsf{attribute\text{-}fault}, P)$ (where $P \in \mathcal{U} \cup \{Op\}$). In this case we say that Γ *attributed a fault to* P. We require that the probability that Γ attributes a fault to an honest party is negligible in λ. An attack $(\mathcal{A}, \mathcal{Z})$ *has no attributable faults* if the probability that Γ attributes a fault to some party is negligible.

2.1 Protocol operation

Let us now describe the general scheme in which a Plasma payment protocol operates. In this section, we focus only on describing what messages are sent between the parties. The "semantics" of these messages, and the security properties of the protocol are described in Sect. 2.2. We assume that all the parties are connected by authenticated and secret communication channels, and a message sent by a party P in the ith round, arrives to P' at the beginning of the $(i + 1)$th round. The communication is synchronous and happens in *rounds*. It consists of three stages, namely: "setup", "initialization", and "payments". The execution starts with the *setup*

[13] We assume that these machines are deterministic, since all their internal randomness will be passed to them by Ψ.

stage. In this stage parameter 1^λ is passed to all the machines in Π. Upon receiving this parameter, machine Ψ samples a tuple $(\psi_{U_1}, \ldots \psi_{U_n}, \psi_{Op}, \psi_\Gamma)$ (where each $\psi_P \in \{0,1\}^*$). Then for each $P \in \{U_1, \ldots, U_n, Op, \Gamma\}$ the string ψ_P is passed to P. Afterwards, the parties proceed to the *initialization stage*. In this stage the environment generates a sequence $(a_1^{\mathsf{init}}, \ldots, a_n^{\mathsf{init}})$ of non-negative integers and passes it to the contract Γ (recall that the state of Γ is public, and hence, as a consequence, all the parties in the system also learn the a_i^{init}'). Then the protocol proceeds to the *payment stage*. This stage consist of an unbounded number of *epochs*. Each ith epoch (for $i = 1, 2, \ldots$) is divided into two *phases*.

Payment phase. In this phase the environment sends a number of *payment orders* to the users (for simplicity we assume that this happens simultaneously in a single round). Each order has a form of a message "(send, v, U_i)", where $v \in \mathbb{Z}_{\geq 0}$, and $U_i \in \mathcal{U}$. It can happen that some users receive no payment orders in a given epoch. It is also ok if a user receives more than one order in an epoch. Informally, the meaning of these messages is as follows: if a user U_j receives a "(send, v, U_i)" message, then she is ordered to transfer v coins to user U_i. We require that this message can only be sent if none of b_i and b_j are equal to $\perp$ (i.e.: if none of U_i and U_j "exited", see below). The parties execute a multiparty sub-protocol. During this executions some of the users send a message "$(\mathsf{received}, v, U_i)$" to the environment $\mathcal{Z}$. This sub-protocol ends when Γ outputs a message $\mathsf{payments\text{-}processed}$.

Exit phase. In this phase the environment sends *exit orders* to some of the users (again: this happens in a single round). Each such order is simply a message "exit". Informally, sending this message to some U_i means that U_i is ordered to exit the system with all her coins. The environment can send an exit message to U_i only if $b_i \neq \perp$ (i.e. U_i has not already "exited", see below). The parties again execute a multiparty protocol. The protocol ends when Γ outputs a sequence

$$\{(\mathsf{exited}, U_{i_j}, v_{i_j})\}_{j=1}^m, \tag{1}$$

where m is some non-negative integer, and each $U_{i_j} \in \mathcal{U}$ and $v_{i_j} \in \mathbb{Z}_{\geq 0}$. For each U_{i_j} in Eq. (1) we say that U_{i_j} *exited (with v_{i_j} coins)*, and we let $b_{i_j} := \perp$. We require that no party can exit more than once. In other words: it cannot happen that two messages $(\mathsf{exited}, U_i, v)$ and $(\mathsf{exited}, U_i, v')$ are issued by Γ.

We make some assumptions on the communication between the parties. Informally we require that if U and U' are some honest users, then the procedure of transferring coins from U to U' is done by a "sub-protocol" involving only parties in the set U and U'. Since we do not have a concept of "sub-protocol" this is formalized as follows:

Communication locality. Two honest users U and U' exchange messages only in epochs in which they do transactions between each other (i.e. a message (send, U, v) is sent by $\mathcal{Z}$ to U', for some v).

This requirement is very natural since Plasma is supposed to work even when an arbitrary set of users is corrupt. Hence, relying on the other users' help in financial

transfers would be impractical. Up to our knowledge all "pure" Plasma proposals in the literature satisfy this requirement. On the other hand: it may *not* hold if we incorporate some techniques that assume some type of cooperation between larger sets of parties (e.g. consensus mechanisms). Examples include: Buterin's Plasma Chash defragmentation (where a large set of users has to regularly cooperate in order to "clean-up" the system), and StarkDEX's "data availability committee" (see Sect. 1.1), if we treat the committee members as "users". One way to view our result is that it implies that such techniques are inherent for every fungible Plasma.

2.2 Security properties

During the interaction with the protocol, the environment keeps track of balances of *honest* users (we do not define balances of dishonest users). Formally, for each honest user U_i it maintains a variable $b_i \in \mathbb{Z}_{\geq 0} \cup \{\perp\}$ (a *balance* of U_i), where the symbol "$\perp$" means that a party exited. It also maintains a variable $t \in \mathbb{Z}_{\geq 0}$ (initially set to 0) that is used to keep track on the amount of coins that have been withdrawn. The rules for maintaining these variables are as follows. Initially, for each $i := 1, \dots, n$ the environment $\mathcal{Z}$ lets $b_i := a_i^{\mathsf{init}}$. Whenever Γ outputs $(\mathsf{exited}, U_i, v)$ (for some U_i and v) we let $b_i := \perp$ and increment t by v. Each time $\mathcal{Z}$ receives a message $(\mathsf{received}, v, U_i)$ from some *honest* U_j, it adds v to b_j (recipient balance) and, if U_i (sender) is honest too, subtracts v from b_i. We require that the environment never issues an order if U_i or U_j exited (i.e. if $b_i = \perp$ or $b_j = \perp$). The environment also never sends an order exit to the same user more than once, and it never sends exit order to a user U_i that already exited (i.e. such that $b_i = \perp$). We have the following security properties.

Responsiveness to "send" orders. Suppose Op, U_i, and U_j are honest, and the environment issued an order (send, v, U_i) to U_j then in the same epoch party U_i sends a message "$(\mathsf{received}, v, U_j)$" to the environment.

Correctness of "received" messages. Suppose U_i and U_j are honest and U_i outputs a message "$(\mathsf{received}, v, U_j)$", then environment has issued an order (send, v, U_i) to U_j in the same epoch.

Responsiveness to "exit" orders. Suppose U_i is honest and the environment issued an order exit to U_i then in the same epoch Γ outputs a message $(\mathsf{exited}, U_i, v)$ (for some v).

No forced exits if operator honest. Suppose Op and U_i are honest and Γ outputs message $(\mathsf{exited}, U_i, v)$ at epoch r, then environment has sent the order exit to U_i in the same epoch.

Fairness for the users. If Γ outputs a message $(\mathsf{exited}, U_i, v)$ (for some honest U_i) then $v \geq b_i$ (where b_i is the current balance of U_i).

Limited responsibility of the operator. If the operator is honest, then the total amount of coins that are withdrawn from the system is at most $a_1^{\mathsf{init}} + \cdots + a_n^{\mathsf{init}}$. Otherwise (if she is dishonest) the total amount of coins that are withdrawn from the system is at most $\lceil (1+\gamma)(a_1^{\mathsf{init}} + \cdots + a_n^{\mathsf{init}}) \rceil$. This definition captures the notion of operator's collateral, and the fact that it is used (to cover users' losses) if the operator is caught cheating.

If an attack $(\mathcal{A}, \mathcal{Z})$ succeeds to violate any of the requirements from this section, then we say that $(\mathcal{A}, \mathcal{Z})$ *broke a given Plasma payment system.* We say that Π is *secure* if for every environment $(\mathcal{A}, \mathcal{Z})$ the probability that $\mathcal{A}$ breaks Π is negligible in 1^λ.

As explained in the introduction, certain attacks on Plasma are of particular importance, due to the fact that they are hard to handle in real life. We say that $(\mathcal{A}, \mathcal{Z})$ *force an on-chain action of size M (in some epoch i)* if the following happened. Let T be the set of honest parties that did *not* receive any order from $\mathcal{Z}$ in epoch i. Then the total length of messages sent by parties from T to Γ is at least M. As explained in the introduction, the term that is more standard than "forced on-chain action" is "mass *exit*". See 1.1 for a discussion why "forced on-chain action" is a better term when impossibility results are considered.

3 Our main result

We now present Thm. 1, which is the main result of this paper. The main implication of this theorem is that for every non-collateralized Plasma system there exists an attack that provokes a mass forced on-chain action, i.e., it forces the honest users to make large communication with the contract even if they did not receive any exit order from the environment (see point 1 in the statement of the theorem), unless a given Plasma system has large exits (point 2). Moreover, this can be done by an attack that has *no* uniquely attributable faults. This fact cannot be circumvented by putting a collateral on the operator, unless this collateral is very large.

Theorem 1 (Mass forced on-chain actions or large exits without uniquely attributable faults are necessary). *Let Π be a secure Plasma payment system with n users and let $\gamma \geq 0$ be the operator's collateral fraction. Then either*

1. *there exists an attack on Π that causes a forced on-chain action of size greater than $(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$ with probability at least $1/16 + \mathsf{negl}(\lambda)$, or*
2. *there exists an attack on Π such that* one *honest user, when ordered to exit by the environment, makes communication to Γ of size at least $(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$ with probability at least $1/16 + \mathsf{negl}(\lambda)$.*

Moreover, both attacks have no uniquely attributable faults.

One way to look at this theorem is as follows. First, consider a non-collateralized Plasma, i.e., assume that $\gamma = 0$. Let $\mathcal{P}^1$ be a class of non-collateralized Plasma that with overwhelming probability *do not* have uniquely attributable forced on-chain actions (of any size larger than 0). In this case point 1 cannot hold, and hence, every Plasma $\Pi \in \mathcal{P}^1$ needs to satisfy point 2. This means that there exists an attack on every $\Pi \in \mathcal{P}^1$ such that one honest user, when ordered to exit by the environment, makes communication to Γ of size at least $(n - 5)/4$ with probability around $1/16$. Or, in other words: every Plasma from class $\mathcal{P}^1$ must have a large exist size with noticeable probability. We know Plasma with such properties: it is essentially Plasma Cash (see the full version of this paper [11])

On the other hand, let $\mathcal{P}^2$ be a class of non-collateralized Plasmas that with high probability have no large exits, in the sense of point 2 of Thm. 1. This means that point 1 has to hold, which implies that every $\Pi \in \mathcal{P}^2$ needs to have large (at least around $(n-5)/4$) non-uniquely attributable mass forced on-chain actions. Plasma with such properties is called Fungible Plasma (see the full version of this paper [11]) Hence, informally speaking, Thm. 1 states that we cannot have the "best of two types of Plasma" simultaneously.

If we consider non-zero collaterals, i.e., we let $\gamma > 0$ then the situation does not improve much, unless the collateral fraction is large, i.e., the total collateral blocked by the operator is at least around $n \cdot \gamma = n/\log_2 n$[14]. This essentially means that we cannot get around the bounds from Thm. 1 by introducing collateral, unless the amount of coins blocked in operator's collateral is of roughly the same order as the total amount of coins stored by the users.

Note that even trivial versions of Plasma "fit" into Thm. 1. For example, consider Plasma in which the operator always puts all the transactions on-chain. Of course, the details would need to be worked out, but clearly such a Plasma can be made secure. The existence of such a trivial Plasma does not contradict our Thm. 1, since it clearly satisfies point 1: a large number of transfers in one epoch will cause a forced mass on-chain action (by the operator). The same also holds if every user needs to put each transaction on-chain.

4 Proof of Thm. 1

Before we present the proof let us introduce some auxiliary machinery. This is done in the next section.

4.1 Isolation scenario

Let Π be a Plasma payment system, let $\mathcal{Z}$ be an environment, and let $\mathcal{W}$ be some subset of the users of Π. We now introduce a procedure that we call *isolation of* $\mathcal{W}$. In this scenario Π is executed as in the normal execution, except that we "isolate" the users $\mathcal{W} \subseteq \mathcal{U}$ from the operator. More precisely: all the messages sent between any $U \in \mathcal{W}$ and the operator Op are dropped, i.e., they never arrive to the destination. This scenario can be viewed as an "attack" although it does not fit into the framework from Sect. 2.1, since it violates the assumption that messages sent by an honest party to another honest party always arrive to the destination.

Although the isolation scenario cannot be performed within our model, it can be "emulated" by corrupting either the operator Op, or the users from $\mathcal{W}$. In the first case we corrupt the operator and instruct him to behave as if she was honest, except that she does not send messages to the users in $\mathcal{W}$ and ignores all messages sent by these users. This will be called the *data unavailability (DU) attack against* $\mathcal{W}$ *by the operator*. In the second, symmetric case (the *pretended data unavailability*

[14] This is because we need to have $\gamma \approx 1/\log_2 n$ to make the expression "$(n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4$" equal to 0.

(PDU) attack by $\mathcal{W}$ on the operator) we corrupt the users in $\mathcal{W}$. Then, every user $U \in \mathcal{W}$ behaves as if she was honest, except that she does not send messages to Op, and ignores all messages from Op.

If this is the only type of malicious behavior, then "from the point of view" of all the other parties, and, most importantly, from the point of view of the contract machine Γ, it is impossible to say who is corrupt (the users in $\mathcal{W}$ or the operator Op). More precisely, we have the following.

Observation 1 *Let Π be a Plasma payment system and consider the attack that isolates users in some set $\mathcal{W}$ from the operator. Let $\mathcal{Z}$ be an arbitrary environment and let $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{isolate}}$ be the random variable denoting the transcript of messages received by Γ. Moreover, let $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{PDU}}$ and $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{DU}}$ be the random variable denoting the transcripts of messages received by Γ in the PDU attack and in the DU attack (respectively), both with environment $\mathcal{Z}$. Then $\mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{DU}} \overset{d}{=} \mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{isolate}} \overset{d}{=} \mathcal{T}^{\mathcal{W},\mathcal{Z}}_{\text{PDU}}$.*

This fact is useful in the proof of the following simple lemma.

Lemma 1 *Fix an arbitrary Plasma Π. Let $\mathcal{W}$ be some set of users. Suppose $\mathcal{A}$ performs a DU attack against $\mathcal{W}$ or a PDU attack by $\mathcal{W}$ (either by corrupting the operator or by corrupting the users), and let $\mathcal{Z}$ be arbitrary. Then the attack $(\mathcal{A}, \mathcal{Z})$ has no uniquely attributable faults.*

Proof. From the security of Π we get that if the users are corrupt then the probability that Γ attributes a fault to them is negligible. Symmetrically, if the operator is corrupt then the probability that Γ attributes a fault to her is negligible. By Observation 1 the transcripts of messages received by Γ in both attacks are distributed identically, so the probability that Γ attributes *any* fault has to be negligible.

4.2 Proof overview

Fix some secure Plasma payment system Π that works for n users. We construct either an attack such that

$$\Pr\left[\begin{array}{c}\text{the set of all honest users makes communication to } \Gamma \\ \text{of size at least } (n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4 \\ \text{(without receiving an } \mathsf{exit} \text{ order from the} \\ \text{environment)}\end{array}\right] \geq 1/16 + \mathsf{negl}(\lambda), \quad (2)$$

or an attack such that

$$\Pr\left[\begin{array}{c}\text{user } U_1\text{, when ordered to exit by the environment,} \\ \text{makes communication to } \Gamma \text{ of size at least} \\ (n - \lceil \gamma n \rceil \cdot \log_2 n - 5)/4\end{array}\right] \geq 1/16 + \mathsf{negl}(\lambda). \quad (3)$$

In both of these attacks the amount of coins given to the users is n, but our proof can be generalized to cover cases when it is required that the amount of coins is larger than n (we comment more on this at the end of Sect. 4). On the other

hand, the proof does not go through in the (unrealistic) case when this amount is very small (sublinear in n).

The attacks that we construct in both cases ((2) and (3)) have no uniquely attributable faults. Note that for $n \leq 5$ Eq. (3) holds trivially, and therefore we can assume that $n > 5$. Let Υ denote the family of all *non-empty proper* subsets of $\{U_2, \ldots, U_n\}$, i.e. sets $\mathcal{V}$ such that $\emptyset \subsetneq \mathcal{V} \subsetneq \{U_2, \ldots, U_n\}$ (note that $U_1 \notin \mathcal{V}$). Since we assumed that $n > 5$ we have that $\log|\Upsilon| = \log_2(2^{n-1} - 2) \geq n - 2$, and, in particular, Υ is non-empty. In the proof we construct an experiment (denoted by $\mathsf{Exp}(\mathcal{V})$ and presented in details in the full version of this paper [11]) and analyze its performance, assuming that $\mathcal{V}$ is sampled uniformly at random from Υ. Depending on this analysis, the experiment $\mathsf{Exp}(\mathcal{V})$ can be "transformed" into an attack that satisfies Eq. (2) or Eq. (3).

Experiment $\mathsf{Exp}(\mathcal{V})$ "simulates" an execution of two epochs of Plasma Π. In the first epoch the adversary isolates the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ from the operator (in the attacks that we construct later this will be done either by corrupting these users, or the operator). The environment gives 1 coin to each user $U \in \mathcal{U}$. Then, in the "payment" phase of the first epoch all the users from $\mathcal{V}$ transfer their coins to U_1. In the "exit" phase of the first epoch user U_1 receives an exit order from the environment and consequently exits with all her coins. Note that in the first epoch every party behaved honestly (except of the isolation attack against the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$), and hence U_1 is guaranteed to successfully exit with her coins (she has 1 such coin from the "initialization" phase, and $|\mathcal{V}|$ coins that were transferred to her by the users in $\mathcal{V}$).

Of course the honest parties from $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ will usually realize that they are isolated from the operator. As a reaction to this they may send some messages to Γ. This, in turn can provoke the other parties to react by sending their messages to Γ. Hence, in general there can be a longer interaction between all the parties and Γ in this phase. Let $\mathcal{T}^1$ be the transcript of the messages sent by the users in $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$ to Γ in both phases, let $\mathcal{T}^2$ be the messages sent by the users in $\mathcal{V}$ and the operator to Γ in both phases, let $\mathcal{T}^3$ be the messages sent by U_1 to Γ in the "payment" phase, and finally let $\mathcal{T}^4$ be the messages sent by U_1 to Γ in the "exit" phase. The first epoch of the experiment $\mathsf{Exp}(\mathcal{V})$ and the transcripts are depicted on Fig. 1.

Before discussing the second epoch of the experiment, let us note that in the first epoch the only way in which we deviate from the totally honest execution is the "isolation" of $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$. This will later allow us to be "flexible" and corrupt different sets of parties ($\{Op\}$ or $\{U_2, \ldots, U_n\} \setminus \mathcal{V}$) depending on the results of our analysis of $\mathsf{Exp}(\mathcal{V})$. This will be different in the second epoch, where we always assume that parties from $\mathcal{V}$ are corrupt. This is ok because while constructing the attacks that satisfy (2) or (3) we will only use the first epoch of $\mathsf{Exp}(\mathcal{V})$. The only reason to have the second epoch of $\mathsf{Exp}(\mathcal{V})$ is to make sure that the users have to send large amounts of data to Γ during the first epoch, as otherwise corrupt $\mathcal{V}$ can steal the money (in the second epoch).

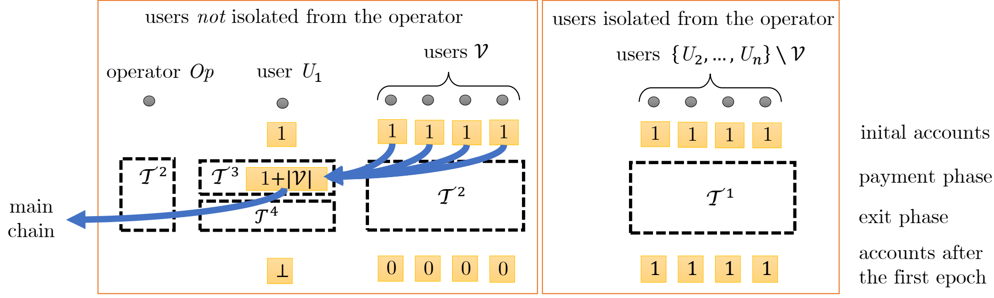

Fig. 1: The first epoch of the experiment $\mathsf{Exp}(\mathcal{V})$. Gray circles denote the parties, and the $\mathcal{T}^i$'s denote the transcripts of the communication with Γ (see, e.g., Sect. 4.2 for their definitions).

Let us now present some more details of the second epoch. Initially we corrupt all the users from $\mathcal{V}$ and "rewind" them to the state that they had at the beginning of the first epoch. This is done in order to let them "pretend" that they still have their coins. We then let all of them try to ("illegally") exit with these coins. Technically, "rewinding a user U" is done via a procedure denoted $\mathsf{Reconstruct}_U$. This procedure outputs the state that U would have at the end of the "payment" phase if she did not transfer her coins to U_1. To make it look consistent with the state of Γ, this procedure takes as input the transcripts defined above. Then, each user $U \in \mathcal{V}$ tries to exit (in the "exit" phase) from her state computed by $\mathsf{Reconstruct}_U$. Also the honest users try to exit (they receive an "exit" order from the environment). Let $\mathcal{Q}$ be the set of users that managed to exit with at least 1 coin. From the security of Plasma we get that $\mathcal{Q}$ is equal to the set of honest users ($\{U_2, \dots, U_n\} \setminus \mathcal{V}$) plus a small (of size at most $\lceil \gamma n \rceil$) subset $\mathcal{D}$ of dishonest users.

The key observation is now that all that is needed to "simulate" the second epoch of $\mathsf{Exp}(\mathcal{V})$ are the transcripts $\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3$, and $\mathcal{T}^4$. On the other hand $\mathcal{V}$ can be approximately computed from $\mathcal{Q}$ (i.e., we can compute $\mathcal{V}$ with elements $\mathcal{D}$ missing, where $|\mathcal{D}| = \lceil \gamma n \rceil$). Hence the variable $(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)$ carries enough information to "approximately" describe $\mathcal{V}$. Thanks to this we can construct a "compression" algorithm that "compresses" a random $\mathcal{V} \leftarrow\$ \Upsilon$ by simulating the first epoch of $\mathsf{Exp}(\mathcal{V})$ and obtaining $(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)$ and then "decompresses" it by simulating the second epoch, and computing the output as $\mathcal{V} := \{U_2, \dots, U_n\} \setminus \mathcal{Q}$ (the additional $\lceil \gamma n \rceil$ elements can be simply listed as an additional output of C and passed to D as input that has to be added to the output of D).

On the other hand, clearly (for completeness we show this fact in the full version of this paper [11]), a random $\mathcal{V} \leftarrow\$ \Upsilon$ with high probability cannot be compressed to a string that is significantly shorter than $\log |\Upsilon| \geq n - 2$. This implies that with a noticeable probability $|(\mathcal{T}^1, \mathcal{T}^2, \mathcal{T}^3, \mathcal{T}^4)| \approx n - \lceil \gamma n \rceil \log_2 n$, where $\lceil \gamma n \rceil \log_2 n$ is the number of bits needed to describe set $\mathcal{D}$.

Obviously, the above fact implies that for at least one $i \in \{1, \dots, 4\}$ we have that $\mathcal{T}^i \geq (n - \lceil \gamma n \rceil \log_2 n)/4$ with noticeable probability for concrete parameters). The rest of the proof of Thm. 1 is based on the case analysis of the implications of

"$\mathcal{T}^i \geq n/4$" for different i's. More concretely, we show that in the first three cases ($i = 1, 2$, and 3) we can construct attacks that satisfy Eq. (2), and in case $i = 4$ — an attack that satisfies Eq. (3). All these attacks are based on the experiment $\mathsf{Exp}(\mathcal{V})$, but are only using its first epoch. In the proof we exploit the fact that the only malicious behavior that happens in this epoch is the "isolation" (i.e., not sending messages). Hence, we can use Observation 1 and "switch" between scenarios when different groups of parties are corrupt (while still getting the same transcripts $\mathcal{T}^i$). Moreover these attacks do not have uniquely attributable faults.

The detailed proof of Thm. 1 can be found in the full version of this paper [11].

Remark 1. Our proof would also go through even if the total balance of the users a is arbitrarily large. The only difference would be that instead of giving 1 coin to every user, the environment would give to each user U_i (for $i > 1$) $\lfloor a/n \rfloor$ coins, and to user U_1 the environment would give the remaining coins (say). The rest of the proof would be essentially identical to the proof of Thm. 1.

Remark 2. Although the attack presented requires two epochs, the second epoch only captures the scenario where the underlying protocol is insecure and hence it can be seen as a "thought experiment". In other words, if the honest parties do not make large communication with the contact Γ in the first epoch, they risk loosing their coins in the second epoch. Therefore, under the assumption that the plasma system is indeed secure and consequently parties make large communication with Γ in the first epoch, the adversary cannot steal any coins in the second epoch and hence the second epoch would become obsolete.

Conclusion

The main contribution of this work is that we have shown that the distinction between Plasma Cash and Fungible Plasma is inherent, i.e., we ruled out the possibility of constructing Plasma that combines benefits of both Plasmas. We believe that, besides of the general scientific interest, our work (especially ruling out existence of some Plasma constructions) can help the practical blockchain community in developing Plasma protocols, and in general can bringing more understanding in what is possible and what is impossible in the area of off-chain protocols, and under what assumptions. It can also serve as a formal justification why "hybrid" approaches (such a "rollups") [6] may be needed in real life. We also hope that this work may expand the scope of theory by identifying a new area where theoretical lower bounds can have direct impact on the real life problems.

References

[1] N. Asokan et al. "Optimistic Protocols for Fair Exchange". In: *CCS '97, Proceedings of the 4th ACM Conference on Computer and Communications Security, Zurich, Switzerland, April 1-4, 1997*. 1997.

[2] V. Buterin. *A Note on Limits on Incentive Compatibility and Griefing Factors.*

[3] V. Buterin. *Minimal Viable Plasma*. 2018.
[4] V. Buterin. *Plasma Cash: Plasma with much less per-user data checking*. 2018.
[5] V. Buterin. *Scalability, Part 2: Hypercubes*.
[6] V. Buterin. *The Dawn of Hybrid Layer 2 Protocols*. `https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html`. (Accessed on 02/08/2020). 2019.
[7] C. Cachin and J. Camenisch. "Optimistic Fair Secure Computation". In: *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*. 2000.
[8] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*. 2001.
[9] S. Dziembowski et al. "FairSwap: How To Fairly Exchange Digital Goods". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018.
[10] S. Dziembowski et al. "General State Channel Networks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018.
[11] S. Dziembowski et al. *Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma*. Cryptology ePrint Archive, Report 2020/175. `https://eprint.iacr.org/2020/175`. 2020.
[12] S. Dziembowski et al. "Multi-party Virtual State Channels". In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. 2019.
[13] G. Fuchsbauer et al. "Aggregate Cash Systems: A Cryptographic Investigation of Mimblewimble". In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. 2019.
[14] J. A. Garay et al. "The Bitcoin Backbone Protocol: Analysis and Applications". In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. 2015.
[15] L. Goldberg and O. Katz. *StarkDEX Deep Dive: Contracts & Statement - StarkWare - Medium*. `https://medium.com/starkware/tagged/starkdex-specs`. (Accessed on 02/08/2020). 2019.
[16] L. Gudgeon et al. "SoK: Layer-Two Blockchain Protocols". In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers*. 2020.
[17] S. Haber and W. S. Stornetta. "How to Time-Stamp a Digital Document". In: *J. Cryptology* 2 (1991).

[18] R. Impagliazzo and S. Rudich. "Limits on the Provable Consequences of One-Way Permutations". In: *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. 1989.

[19] H. A. Kalodner et al. "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design". In: *14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22-23 June, 2015*. 2015.

[20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. 2007.

[21] R. Khalil et al. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Report 2018/642. `https://eprint.iacr.org/2018/642`. 2018.

[22] A. Kiayias and O. S. T. Litos. "A Composable Security Treatment of the Lightning Network". In: *IACR Cryptology ePrint Archive* (2019).

[23] G. Konstantopoulos. *Plasma Cash: Towards more efficient Plasma constructions*. 2019.

[24] R. Mitra. *Plasma Breakthrough: OmiseGO (OMG) announces the launch of Ari*. `https://www.fxstreet.com/cryptocurrencies/news/plasma-breakthrough-omisego-omg-announces-the-launch-of-ari-201904120245`. (Accessed on 02/08/2020).

[25] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2009.

[26] J. Poon and V. Buterin. *Plasma: Scalable Autonomous Smart Contracts*. 2017.

[27] N. Szabo. *Smart Contracts: Building Blocks for Digital Markets*. Extropy Magazine. 1996.

[28] Trustnodes. *Ethereum Transactions Fall Off the Cliff, Three Plasma Projects Close to Release Says Buterin*. 2018.

[29] Wikipedia. *Trusted timestamping*.

[30] J. I. Wong. *The ethereum network is getting jammed up because people are rushing to buy cartoon cats on its blockchain*. Quartz. 2017.

[31] G. Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. 2016.

E. CommiTee: An Efficient and Secure Commit-Chain Protocol using TEEs

This chapter corresponds to our work on CommiTee, an efficient off-chain Plasma/commit-chain protocol designed using TEEs. Our paper can also be found in [72].

CommiTEE : An Efficient and Secure Commit-Chain Protocol using TEEs

Andreas Erwig[1], Sebastian Faust[1], Siavash Riahi[1], and Tobias Stöckert[1]

[1] Technische Universität Darmstadt, Germany, first.lastname@tu-darmstadt.de
[2] Fraunhofer SIT, Darmstadt, Germany, tobias.stoeckert@sit.fraunhofer.de

Abstract. Permissionless blockchain systems such as Bitcoin or Ethereum are slow and expensive, since transactions are processed in a distributed network by a large set of parties. To improve on these shortcomings, a prominent approach is given by so-called 2nd-layer protocols. In these protocols parties process transactions off-chain directly between each other, thereby drastically reducing the costly and slow interaction with the blockchain. In particular, in the optimistic case, when parties behave honestly, no interaction with the blockchain is needed. One of the most popular off-chain solutions are Plasma protocols (often also called commit-chains). These protocols are orchestrated by a so-called operator that maintains the system and processes transactions between parties. Importantly, the operator is trustless, i.e., even if it is malicious users of the system are guaranteed to not lose funds. To achieve this guarantee, Plasma protocols are highly complex and rely on involved and expensive dispute resolution processes. This has significantly slowed down development and deployment of these systems.
In this work we propose CommiTEE – a simple and efficient Plasma system leveraging the power of trusted execution environments (TEE). Besides its simplicity, our protocol requires minimal interaction with the blockchain, thereby drastically reducing costs and improving efficiency. An additional benefit of our solution is that it allows for switching between operators, in case the main operator goes offline due to system failure, or behaving maliciously. We implemented and evaluated our system over Ethereum and show that it is at least 2 times (and in some cases more than 16 times) cheaper in terms of communication complexity when compared to existing Plasma implementations. Moreover, for protocols using zero-knowledge proofs (like NOCUST-ZKP), CommiTEE decreases the on-chain gas cost by a factor ≈ 19 compared to prior solution.

1 Introduction

Over the past decade cryptocurrencies such as Bitcoin [35] and Ethereum [47] have gained increasing popularity by introducing a new financial paradigm. Unlike traditional financial systems these cryptocurrencies do not rely on a central authority for transaction validation and accounting, but instead build upon a decentralized consensus protocol which maintains a distributed ledger that tracks each single transaction. However, maintaining such a ledger in a distributed fashion comes at the cost of poor transaction throughput and confirmation time. For example, in Ethereum the transaction throughput is limited to a few dozen transactions per second and final confirmation of a transaction can take up to 6 minutes. On the contrary, traditional centralized payment providers offer almost instantaneous transaction confirmation while being able to support orders of magnitude higher throughput. These scalability issues hinder cryptocurrencies from being used at larger scale.

One particularly promising solution to address these scalability problems are off-chain protocols. Off-chain protocols work by taking the massive bulk of transactions

off-chain, and at a high-level proceed as follows. After an initial on-chain transaction to join the system, transactions between participants can be carried out off-chain (without interaction with the underlying blockchain). Only when a user wants to exit the system, or when other parties of the system try to cheat, honest users need to carry out on-chain transactions again. Important examples of this concept include payment channel networks (or hubs) [41, 30, 5], and more recently Plasma protocols, where the latter is the focus of this work.

The original idea of Plasma protocols (often also known as commit-chains[3]) was first introduced by Poon et al. in 2017 [40]. Soon after, countless variants of Plasma emerged such as Loom, Bankex, NOCUST and OmiseGO (see, e.g., [44, 24, 33]). While all these systems differ in certain aspects, at a high-level they all follow the off-chain approach described above and outsource transaction execution to a second layer (we will explain in more detail how Plasma systems work in Sec. 3). A key role in any Plasma system plays the *operator*, who maintains the system, and ensures that all off-chain transactions among the users are processed correctly. It is important to note, however, that the operator is not assumed to be trusted, and merely ensures an efficient and well-functioning system. Since the operator is not trusted, but essentially "confirms" all transaction processing, Plasma protocols use a highly involved mechanism which ensures that the user's funds remain secure even when the operator is malicious.

The design of such mechanism poses non-trivial challenges that existing Plasma systems attempt to solve by employing either heavy cryptographic machinery such as zero-knowledge proofs or complex challenge-response protocols for resolving disputes on-chain. Neither of these approaches is optimal for the following two reasons: (1) both approaches significantly increase the communication complexity with the blockchain which increases costs and undermines the original purpose of Plasma as an off-chain protocol; (2) the security analysis of the resulting protocols becomes cumbersome, and hence to date there is no Plasma-like system that has been formally proven secure. While there have been significant research efforts to address these problems [51], [39], [19], the community has not yet come up with a suitable solution that can readily be implemented.

1.1 Our Contribution

In this paper, we address the before mentioned shortcomings.

Secure and Efficient Plasma with TEE We first propose a general security model for Plasma systems, and introduce CommiTEE – a simple and efficient Plasma-like protocol. CommiTEE leverages trusted execution environments (TEE) in order to overcome the above mentioned downsides of existing Plasma systems. In CommiTEE, the operator uses a TEE for all necessary computation, which limits the operator's role in the protocol to merely relaying messages between the TEE and the outside world (blockchain, users of the system, etc.). This significantly limits the operator's ability to misbehave, which in turn allows for a simpler protocol design.

[3] In this work we use the terms Plasma and commit-chain interchangeably.

An immediate consequence of CommiTEE's simplicity is that it becomes easier to analyze and verify its security. This is in contrast to other Plasma protocols which are often too complex for a thorough security analysis. We analyze CommiTEE in our model and show that it achieves the three security properties *deposit security*, *balance security* and *operator security*, which in a nutshell guarantee that no honest party in the system loses her coins even if *all* other parties are malicious.

Another crucial benefit of our simple protocol design is that CommiTEE requires minimal interaction with the blockchain, which significantly reduces on-chain costs and improves efficiency. Unlike payment channels, all existing Plasma/commit-chain protocols require periodic commitments to the blockchain and logarithmic size messages to withdraw coins from the system. CommiTEE overcomes these two drawbacks of conventional commit-chains, which brings the efficiency on par with payment channel networks/hubs [41, 30, 5].

Evaluation We evaluated our system over Ethereum and compared the results to the two most common (and partially in prototype-status available) Plasma protocols, namely Plasma MVP [9] and Plasma Cash [17], as well as to the most prominent commit-chain protocols NOCUST and NOCUST-ZKP [24]. The evaluation shows that CommiTEE is between 2 to 16 times cheaper in terms of communication complexity when compared to Plasma MVP and Cash. For systems using zero-knowledge proofs (e.g., NOCUST-ZKP), CommiTEE outperforms previous work with respect to on-chain communication by a factor of ≈ 19. Furthermore, asymptotically the on-chain communication complexity of CommiTEE is $O(1)$, while for all the above mentioned protocols the communication complexity grows logarithmically in the number of users.

Extensions to CommiTEE Finally, we propose two extensions for CommiTEE in order to (1) support multiple operators, and (2) handle TEE compromise. The first extension to a multi-operator system allows switching between operators, in case the main operator behaves maliciously or goes offline, which is a highly desirable feature for Plasma systems as it makes them more robust to operator failures, and thus increases their reliability significantly. We emphasize that until now there has not been any full specification of a Plasma-like system that supports multiple operators due to the complexity of designing such a system. This gap is closed by our work leveraging again the power of TEEs. Likewise, our second extension to handle TEE compromise significantly increases the reliability and practicality of CommiTEE as it guarantees the security of user's funds even in case a malicious party is able to influence the computation inside the TEE.

1.2 Applications to Decentralized Finance (DeFi)

Although our focus in this paper is to build an efficient off-chain protocol for monetary transactions, CommiTEE is quite versatile and can be extended to instantiate DeFi applications off-chain. This is mainly due to the usage of TEE which allows us to extend the transaction logic and support additional functionalities. As an example, we can build an off-chain marketplace, where parties can not only have balances

but also own Non-Fungible Tokens (NFTs). These tokens can then be traded and sold for some agreed upon value or withdrawn on-chain for further usage. Furthermore, it is possible to support event-based transactions via so-called oracles (e.g., [12]), i.e., an off-chain transaction only happens if an oracle service announces an event on-chain. Last but not least, it is possible to support loans, e.g., users can stake some token to receive funds or use flash loans where the loan and its interest are issued and returned within a single transaction. Therefore, one can build a fully fledged off-chain DeFi marketplace. For more information regarding such DeFi applications we refer the reader to works such as [2, 46, 42].

1.3 Related Work

We briefly discuss the most important related works on commit-chains and off-chain solutions using TEEs.

Plasma protocols There are many different variants of Plasma protocols. Some of the most well known are Plasma MVP [9], Plasma Cash [17] Plasma Debit [38] and Plasma Snapp [39]. Yet most of these protocols have been mentioned and discussed only in forums, e.g., the `https://ethresear.ch` website and there has not been any academic work that formalizes them. To date, the only formally presented commit-chain/Plasma solutions are the NOCUST and NOCUST-ZKP protocols by Khalil et al. [24], both of which however suffer from the aforementioned drawbacks of requiring either a complex on-chain challenge-response mechanism or zero-knowledge proofs.

Dziembowski et al. [16] give a lower bound for the communication complexity in Plasma protocols, showing that any secure Plasma protocol requires significant communication with the blockchain. Our protocol does not violate this lower bound, but shows how to significantly reduce the concrete communication complexity with the blockchain. As we show in this work such a system can be quite efficient and cheap in terms of blockchain interaction and on-chain computation complexity.

Recently, a new proposal called Rollups has gained more popularity [11]. In this approach, the operator publishes the entire raw transaction data to the blockchain in order to avoid some of the fundamental drawbacks pointed out by Dziembowski et al. [16]. Yet, this solution requires significant modifications to the underlying blockchain in the sense that the blockchain must be able to store large amounts of data temporarily. As pointed out by Buterin [8], it will likely take years to develop these solutions. Our work, however, aims for a solution that works without any modifications to the existing underlying blockchain.

Off-Chain TEE Solutions In a recent work, Das et al. [15] proposed the FastKitten protocol which allows parties to execute arbitrary complex smart contracts off-chain even if the underlying blockchain does not support smart contract execution. To this end, the authors use an operator who has access to a TEE which allows efficient and correct execution of smart contracts off-chain. Yet, the set of parties who participate in the smart contract is fixed for the entire lifetime of the contract execution. In addition, the operator has to be collateralized, i.e., make a security deposit, which is

as large as the initial balance of all users combined. A similar solution to [15] has been proposed by Cheng et al. [13], which considers confidentiality preserving off-chain smart contract executions using a TEE. Similarly, [6] and [23] propose solutions for private off-chain function execution with the help of TEEs. However, the main goal of these works is to move complex contract executions off the chain, yet require the encrypted state of a contract execution to be published on the blockchain after each function call, which results in significant interaction with the blockchain. Our work on the other hand aims at reducing on-chain communication complexity. Two other works [25], [22] present privacy preserving off-chain executions of contracts using TEEs, yet both of them rely on zero-knowledge proofs which we avoid in our solution.

Lind et al. [28] proposed the Teechain in order to improve the transaction throughput of payment channels and payment channel networks. They utilize TEEs in order to process transactions and reduce blockchain interaction in case of disputes. In order to deal with TEE failures or compromises, a committee of TEEs is used who must agree on the latest state of the balances. As this work focuses on payment channel networks, parties that do not have a direct channel must find a path in the network through some intermediaries who must have enough balance in order to facilitate such transaction.

There has been a considerable amount of work on the usage of TEEs in conjunction with a blockchain in order to enhance existing blockchain applications ([50, 49, 3, 48, 31] and many more), which, however, does not focus on off-chain applications and is hence not closely related to our work.

2 Preliminaries

In this section we give a brief overview of the main concepts our protocol depends on, namely blockchain and trusted execution environment.

2.1 Blockchain and Cryptocurrencies

Throughout this paper we model a blockchain $\mathbf{b}$ as a sequence of individual blocks $(b_1, \cdots, b_m)$ where b_m is the latest confirmed block, meaning that it is the latest block that the blockchain network reached consensus on. We use the terms *blockchain* and *ledger* (denoted by $\mathcal{L}$) interchangeably in this paper. Users participating in the blockchain network have an account which stores their current balance in the system. The account of a party P_i is identified by her public key pk_i and it consists of a tuple (pk_i, v_i) where v_i is the current balance of P_i. Furthermore, the blockchain network can execute Turing complete programs referred to as smart contracts. Similar to parties, a smart contract has an account which is identified by a unique address addr, however in contrast to parties, each contract consists of a set of functions $(f_1, \cdots, f_l)$ which might optionally take parameters param as input.

In order to transfer funds from a party P_i to another party P_j, P_i must submit a transaction tx of the form (pk_i, pk_j, v) to the blockchain network. Such a transaction

is valid if it is signed under the public key pk_i, and P_i's balance exceeds v. We assume that publishing a transaction takes at most time Δ.

Modeling Blockchain Validation Any blockchain system inherently requires a validation algorithm that allows to check if a sequence of blocks was computed correctly. This validation includes checks for the validity of transactions in individual blocks and checks that two consecutive blocks form a valid chain. In order to verify the whole blockchain a user has to start the verification process with the very first block (genesis block) of the blockchain. This results in a potentially huge amount of storage and computation cost since the user has to download and store the entire blockchain. Therefore, in practice nodes often verify new blocks with respect to so-called checkpoints instead of the entire blockchain. A checkpoint is a confirmed block on the chain whose validity has been verified before.

2.2 Trusted Execution Environments

Our TEE modeling follows the one of Das et al. [15] and Pass et al. [37]. As in [15], we only explain a simplified version of the model and refer the reader to Figure 1 in [37] for the complete formal definition. When the TEE is initialized it creates a signing keypair (msk, mpk) called master public key and master secret key of the TEE. The master keypair is used to authenticate the installation of a program on the TEE. The TEE functionality offers two enclave operations, an install and a resume operation. The install operation stores a given program p under an enclave identifier eid. A program that is stored on an enclave is executed via the enclave operation resume. It takes the identifier of the enclave eid, a function f, and the function input in as input and returns the output of the program operation denoted as out and a quote ϱ over the tuple (eid, p, out). This quote serves as the verifiable statement in the remote attestation process which allows to verify that out was output by an enclave with master public key mpk that installed program p.

Intel SGX [32, 20] and ARM TrustZone [27] are probably the most well known TEEs that are used in practice. Most recently the introduction of SGX2 [21] tackles some limitations of the original SGX proposal e.g., by increasing its memory capacity to up to 512GBs. Nevertheless, in this work remain TEE-agnostic, and do not limit ourselves to a certain brand. By doing so our protocol can be implemented and deployed on a large variety of hardware that satisfy our modeling.

We note that making TEEs secure against side-channel [45], memory-corruption [4] and architectural vulnerabilities [7] is an ongoing research direction which is orthogonal to our work [1], [45], [43]. In CommiTEE we consider a TEE which is secure against such vulnerabilities. However, in Section 7.2, we propose an extension to CommiTEE, that allows users to detect TEE corruption such that user's funds remain secure even in case a malicious party can influence the computation inside the TEE.

3 Solution Overview

In this section, we provide a brief overview of our solution and discuss its main design challenges that had to be overcome.

Plasma protocols have first been introduced in order to mitigate the issues of payment channel hubs (PCHs), namely the problem that intermediaries in PCHs are required to lock a significant amount of collateral. This is needed in payment channel hubs in order to punish the intermediary in case of malicious behavior. In contrast, Plasma protocols do not utilize a punishment mechanism but instead require the operator to periodically submit a short commitment of the latest state of the system to the blockchain. This avoids the need for collateralization. In case the operator behaves maliciously, users can simply exit the system based on the last commitment.

In a nutshell, users in a Plasma protocol can dynamically join and leave the system by making deposits or exiting their funds from the Plasma contract. They can submit their transactions off-chain to the operator, who collects these transactions and updates the balances of the users accordingly. In addition, the operator stores these transactions and balances in a data structure, which enables her to generate a short commitment to the state of the system. The operator can then publish this commitment on the blockchain and provide a proof of balance to each user, which is required if the user wishes to exit the system. Traditionally, the data structure used by the operator in order to store balances and transactions is a Merkle tree, and the commitment is the root of this tree. In order to exit the system, users must prove to the contract that they own some coins in the system. This proof is in fact a Merkle proof where the leaves store the balances of the users in the system.

As it can be seen, a malicious operator can misbehave by publishing an incorrect commitment (i.e., Merkle root). As an example, the operator can allow users to double spend or "print money" by maliciously increasing the balance of users. In order to mitigate such attacks, existing Plasma protocols employ either of the following two strategies: (1) a complex challenge-response mechanism or (2) a zero-knowledge proof of the operator's correct behavior. These approaches require additional communication or expensive computation on the blockchain, which undermines the original goal of designing an off-chain solution.

Our protocol mitigates the above mentioned issues in two ways: (1) the operator employs a TEE which does all the necessary computation while the operator simply has to relay messages between the TEE and the users and the blockchain and (2) our protocol allows for efficient extension to a multi-operator system, in which a different operator can take over as soon as the current operator acts maliciously.

Since by definition the TEE executes all computations correctly, neither expensive zero-knowledge proofs nor complex challenge-response mechanisms are required in order to guarantee correct behavior of the operator. Instead, a single signature from the TEE is sufficient as a proof of correct computation. In a bit more detail, the proof of balance consists merely of a message signed by the TEE, which is not only much shorter than a Merkle proof but also less expensive to verify on the blockchain. Note that asymptotically the size of a Merkle proof grows logarith-

mically in the number of users, while a signature size is constant. This difference shows itself clearly in Section 6 where we present the evaluation of our protocol. Furthermore, other Plasma systems require periodic commitments of the operator to the blockchain such that users can verify their balance proof. Conventionally, this commitment is a Merkle root. Yet we observe that in order to verify signatures from the TEE, only the TEE's public key is required and hence there is no need for the operator to publish a commitment. Instead, the public key of the TEE is part of the public parameters of the contract.

This efficiency improvement directly translates to the challenge-response mechanism of our protocol. Due to the use of a TEE, many of the scenarios where the operator can act maliciously are mitigated (e.g., changing balances, double spending etc.) and hence the only possible way for the operator to misbehave is by withholding data, i.e., balances signed by the TEE. Due to this limited attack vector, we only need a simple challenge-response mechanism to deal with data unavailability. Note that even this challenge-response mechanism is more efficient compared to traditional Plasma systems, since only a signature needs to be published on the blockchain (instead of a Merkle proof).

3.1 Design Challenges

In the following, we describe the design challenges for Plasma protocols. Even when utilizing a TEE the operator acts as a relay between the TEE and the Plasma system and hence can still act maliciously by dropping the messages sent to and from the TEE. The aim of our protocol is to ensure security even in presence of a malicious operator.

Malicious Operator Since the operator runs the TEE, she controls all interactions with it. This implies that she may send false requests to the enclave or refuse to forward requests made by users. The operator may also abort executions on the enclave or delay inputs and outputs.

At the very beginning, an operator can set up the TEE in a malicious way which would compromise the Plasma system right from the start. Hence, before joining the Plasma network, it is crucial that the users verify the correct initialization of the TEE via remote attestation. This ensures that the correct program has been loaded into the enclave and hence the initialization has indeed been correct.

Since an operator may go offline, crash or act maliciously at any point, users must always be able to withdraw their coins from the system. To this end all users receive a message signed by the TEE which informally says "this user owns v coins in epoch e[4]" and can be sent to the ledger $\mathcal{L}$ if they wish to exit. However, a malicious operator can avoid delivering this message to the users. Yet, the fact that these users did not receive this message from the operator does not have a digital footprint and hence they cannot prove to the contract that the operator misbehaved. Even worse the operator can avoid forwarding the messages produced by the TEE to only a

[4] An epoch is a time interval where the duration is defined by a fixed number of blocks.

subset of users. This would fragment the system into users who have the latest state of the system and users who do not. This means that from the point of view of some users the balances are updated yet for others this is not the case. Since users cannot have two different balances at the same time, our solution must provide a method in order to solve this fragmentation.

A natural question would be whether or not one can extend Plasma protocols in order to support multiple operators and mitigate the single point of failure that arises from having only a single operator. Unfortunately, this is not possible in a straightforward way since all operators must be aware of the latest state of the system and agree on it. This would require them to run a consensus algorithm which, however, significantly hinders the efficiency of the system. In our work, we present an extension to CommiTEE for multi-operator support that does not rely on consensus among the operators and requires only a single on-chain transaction in the best case.

Blockchain Verification In order to process deposits or exits that happen on-chain, the TEE must be informed about the latest state of the ledger every epoch. Hence, we need a secure blockchain verification algorithm on the TEE to ensure that the operator cannot provide incorrect or tampered blocks. We tackle this challenge similar to [15] in the following way.

In order to verify that a deposit (or exit) has been included in the smart contract, the block in which the deposit list is stored, has to be confirmed k-times on-chain. The parameter k is part of the security parameter in the enclave. This requirement ensures that it is computationally infeasible for a malicious operator to forge a valid chain of blocks which prevents potentially malicious deposit attempts by the operator.[5]

Furthermore, in order to make the verification algorithm more efficient, the TEE uses checkpoints in order to avoid validating the entire blockchain each time.

Minimizing Blockchain Interaction Since interacting with the blockchain is slow and expensive, it is crucial to minimize all communication with the blockchain. In Plasma protocols, the operator has to prove at the end of every epoch to all users that she processed all transactions correctly. This is often done by requiring the operator to provide a zero-knowledge proof on-chain such that every user can verify the operator's computation or by allowing users to challenge the operator's behavior on-chain. In our protocol, we make use of the fact that *all* computation done in the TEE is by definition correct. As mentioned before a user's exit message is of the form "this user owns v coins in epoch e" which is signed by the TEE. As we can see, a single signature verification is sufficient in order to verify this statement and no additional Merkle proof or zero-knowledge proof validation is required. In other words, the only on-chain communication happens when a party wishes to join (deposit) or leave (exit) the system[6].

[5] Additionally, this ensures that the block containing the deposit list is not part of a fork on the blockchain.
[6] Note that in Ethereum contracts do not activate on their own and in order to indicate that an epoch or phase has ended, a single transaction must be sent to the contract.

Mass actions and exit size In case the operator is behaving maliciously by performing a data unavailability attack and not responding (or responding incorrectly) to on-chain challenges, users have to exit the system. This often requires the need for mass actions, where all users exit the system at the same time. Such mass actions can result in huge on-chain communication complexity, network congestion and elevated transaction fees. In CommiTEE we do not remove the possibility for mass actions, but the users do not need to exit immediately, thereby circumventing the drawbacks that arise from mass exits as mentioned above.

4 The Plasma Framework Model

In a nutshell, a Plasma protocol Π is a protocol executed between a set of users $\mathcal{P}$, an operator $\mathcal{O}$ and the ledger $\mathcal{L}$ which executes the Plasma contract. The execution of the Plasma protocol consists essentially of three phases, i.e., *deposit*, *transaction* and *exit* phase. In the *deposit phase* users can deposit coins on the ledger into the Plasma contract, in the *transaction phase* users can transfer coins (off-chain) to one another and in the *exit phase* users can withdraw coins from the Plasma contract on-chain. These phases are executed in order and after the exit phase the protocol continues by executing the deposit phase again. Each consecutive execution of these three phases is referred to as an *epoch*. More formally, the following messages are exchanged during each epoch:

Deposit phase In this phase any user P_i can send a message of the form $(\mathsf{deposit}, v)$ to the contract, where v denotes the amount of coins that the user wants to deposit into the Plasma contract. P_i eventually outputs $(\mathsf{deposited}, v', P_i)$ where either $v' = v$ if the deposit was successful or $v' = 0$ otherwise. If the deposit was successful and $P_i \notin \mathcal{P}$, then set $\mathcal{P} = \mathcal{P} \cup \{P_i\}$.

Transaction phase In this phase each user $P_i \in \mathcal{P}$ can send a message of the form (P_i, P_j, v) to the operator $\mathcal{O}$. This message indicates that P_i wants to send v coins to user P_j. At the end of this phase each user P_i receives a message $(v, epoch_{\mathsf{E}}, P_i, \pi)$ from the operator which indicates its balance v in the current epoch $epoch_{\mathsf{E}}$ and includes a balance proof π. A protocol might (optionally) require $\mathcal{O}$ to send the message $(\mathsf{submit}, commit_e)$ to the contract.[7] The contract eventually outputs a message $m \in \{\mathsf{success}, \mathsf{failed}\}$.

Intuitively, a transaction is valid if the sender owns more coins than the transaction amount and the transaction is authorized by the sender (which is commonly checked using digital signatures). The operator is responsible for processing the transactions and updating the balances of the users.

Exit phase In this phase each user $P_i \in \mathcal{P}$ can send a message of the form (exit) to the contract. The contract eventually outputs $(\mathsf{exited}, P_i, v)$, where v denotes the user's latest balance in the Plasma system. P_i is then removed from $\mathcal{P}$, i.e., $\mathcal{P} = \mathcal{P} \setminus \{P_i\}$.

[7] $commit_e$ is a commitment to the updated balances in epoch e, e.g., it can be a Merkle root or a zero-knowledge proof.

4.1 Communication and adversarial assumptions

We now discuss the communication and adversarial assumptions in our modeling. In our model, all parties have access to a ledger $\mathcal{L}$ which supports the execution of Turing complete programs as described in Section 2.1.

A Plasma protocol is executed in presence of an adversary who can corrupt parties. The corrupted parties are "controlled" by the adversary and can deviate from the protocol description. Furthermore, the adversary can delay messages sent by honest users to the ledger.

All parties are connected via authenticated channels i.e., the adversary can read the messages sent between parties and can drop them, yet the adversary is not able to modify the messages that are being sent.

4.2 Properties

We now discuss the properties that a Plasma protocol must fulfill. These properties can be divided into three categories, namely *correctness*, *security* and *efficiency*. The correctness properties describe the protocol's behavior in the optimistic case where parties behave honestly and the security properties describe what the protocol guarantees to honest parties in the pessimistic case, i.e., in presence of malicious parties. We provide in this section only an informal description of these properties. For a formalization of the properties we refer the reader to Appendix B.

Deposit and Transaction Phase Correctness During the deposit phase if an honest user successfully deposits v coins in the Plasma contract and the operator is honest, the balance of this user in the Plasma system is increased by v. During the transaction phase if the sender and the receiver of a transaction and the operator are honest, a transaction of the form (P_i, P_j, v') is processed correctly if the balance of the sender P_i is decreased by the amount v' and the receiver's balance is increased by v'. We note that a user might send/receive coins to/from multiple users during this phase. Balances do not have to be updated immediately but only at the end of the transaction phase. This feature is usually referred to as *late finality* or *eventual finality*.

Exit Phase Correctness If an honest user exits the Plasma system, her balance and the user set $\mathcal{P}$ are updated accordingly. For simplicity we assume that a user always exits with all her coins, and hence her balance is set to 0 and the user is removed from the Plasma user set.

Deposit Security In presence of malicious parties, an honest user does not lose the coins she deposited. In other words, an honest user is able to get her deposits back if they are not processed correctly and her balance is not updated accordingly.

Balance Security In presence of malicious parties, an honest user does not lose any coins at any stage of the protocol, i.e., an honest user is able to always exit her entire balance. We note that due to the late finality feature, this property essentially states

that users will either be able to exit with their balance from the previous or current epoch.

Operator Balance Security An honest operator does not lose the coins she deposited in the Plasma contract, even in presence of malicious users.

Efficiency Let δ denote the duration of an epoch. A Plasma protocol is efficient if $\delta \in O(1)$, i.e., the duration of an epoch is independent of the number of users or transactions.

We note that in practice the duration of an epoch can be dynamically changed based on the number of transactions that are received. For instance, if none of the users issues any transaction, the duration of an epoch can be extended. However, asymptotically the duration of an epoch will always be constant.

5 CommiTEE Protocol

We now give a description of our Plasma protocol which makes use of the fact that the operator runs a TEE. We first give an overview of the protocol execution, before presenting the CommiTEE protocol and the corresponding enclave program that is run by the operator's TEE. Lastly, we provide a high level discussion about the security of our protocol.

5.1 Architecture and Protocol Overview

On a high level, the execution of CommiTEE can be separated into the following phases: (1) Initialization of the system, (2) Verification of the TEE, (3) Depositing coins into the system, (4) Transferring coins off-chain between the participants and (5) Exiting the system (see Figure 1). Note that the phases (3), (4) and (5) occur repeatedly. The execution of the protocol starts with $\mathcal{O}$ initializing the TEE by installing the program p_{CT} and calling its genKeys function. This function will initialize all the necessary variables and generates the key pair $(sk_{\mathsf{E}}, pk_{\mathsf{E}})$. The public key pk_{E} is hard coded on the contract Γ and is used to verify messages which are signed by the TEE. At this point the contract Γ is deployed on the blockchain and users can join the system by depositing coins on-chain on the contract. However, before joining, users first verify that the TEE and Γ are initialized correctly. After a successful verification users can deposit coins on Γ which will increase their balance off-chain in the system. Users who have already deposited some coins can make off-chain transactions by submitting their transactions to $\mathcal{O}$. The operator gathers these transactions and submits them to the TEE, which then updates the balances of the users according to the issued transactions. Furthermore, the TEE signs the updated balances of the users and outputs the list of signed balances which the operator forwards to the users. This signed message can be used as evidence of user's balance, if a party wishes to exit the system. Finally, users who wish to leave the system can submit the signed balance to Γ.

However, if the users did not receive their signed balance from $\mathcal{O}$, they will not be able to exit the system. To mitigate this, users can request an exit (called exit challenge) on Γ. This challenge essentially forces the operator to submit the signed balance to the blockchain. If the request is correctly responded to, the user exits normally with her latest balance. However, if the operator continues to behave maliciously and resists responding to the challenges, the affected users do not have any possibility of exiting their latest balance. In this case Γ will deem $\mathcal{O}$ as malicious and all users must exit according to their balance from the previous epoch. We note that all users have indeed received their signed balance from the previous epoch, otherwise they would have challenged the operator during the exit phase of the previous epoch.

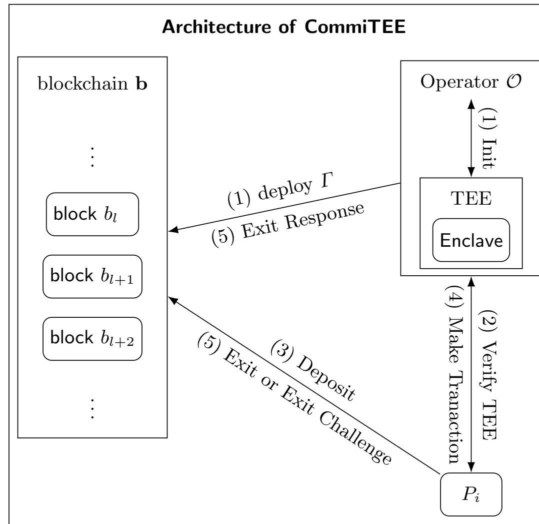

Fig. 1: The architecture of CommiTEE.

5.2 Protocol Description

We now give the description of CommiTEE. We present the corresponding pseudocode of the protocol in Figure 2.

Notation We refer to the protected memory in the operator's TEE where the CommiTEE program is installed and executed as enclave which we denote as E. For simplicity, from this point on, we use the terms enclave and TEE interchangeably. In the rest of this work we use the value $epoch_X$ which denotes the current epoch counter stored by party $X \in \mathcal{P} \cup \{\mathcal{O}\} \cup \{\mathsf{E}\} \cup \{\Gamma\}$. The term epoch refers to the duration of executing the phases (3), (4) and (5). In our protocol an epoch has a fixed duration which is measured by the number of blocks produced on the blockchain. Furthermore each phase of the protocol has a fixed duration. For simplicity we assume that this duration is the same for all three phases and is equal to the time

required to publish t blocks. When we say that a tuple of values is valid, we mean that the tuple has been created according to an honest protocol execution.

Initialization The first step of our protocol is the initialization of the TEE by the operator $\mathcal{O}$. In this stage the TEE first creates its master (signing) key pair (msk, mpk) and outputs the master public key mpk. Then, the operator installs the enclave program p_{CT} with the parameters (κ, Γ, b_{cp}) on the TEE, where κ is the security parameter, Γ is the address of the Plasma contract on $\mathcal{L}$ and b_{cp} is the latest block on $\mathcal{L}$. This block acts as checkpoint for the verification of future blocks. In order to prevent $\mathcal{O}$ from submitting forged blocks or unconfirmed blocks to the enclave, the enclave also derives the parameter k from the security parameter κ. Essentially, the enclave only considers a block b_l as confirmed if it receives a valid chain of blocks $\bar{\mathbf{b}} := (b_l, \cdots, b_{l+k})$.

Upon the completion of the initialization step, the operator calls the key generation function in the enclave, in order to create a second key pair and to initialize internal values specific to the program p_{CT}. The TEE returns $((pk_{\mathsf{E}}, \sigma), \varrho)$ where $(pk_{\mathsf{E}}, \sigma)$ is the generated public key and σ is a signature for this public key under the TEE's master secret key msk. Finally, the value ϱ is the quote from the enclave which allows other users to verify the correct initialization of the installed program. We note that the public key pk_{E} is hard coded on the contract Γ (this can be done after the TEE is initialized by calling a function and fixing pk_{E}). The generation of the additional key pair guarantees that even if a malicious $\mathcal{O}$ re-installs the program on the TEE, Γ will reject messages signed under the new key pair and therefore $\mathcal{O}$ cannot reset the system or revert the balances of the users to 0. This is because the key generation algorithm is probabilistic and will not generate the same key pair except with negligible probability.

Verifying the Enclave Before joining the Plasma system, the users must be convinced that the correct program is installed on the TEE of the operator and that it is registered with the Plasma contract. Otherwise the operator might have installed a different program on the TEE which maliciously increases or decreases the balances of the users. To this end, the user verifies the quote ϱ which ensures that the program $p_{CT}(\kappa, \Gamma, b_{cp})$ has been initialized correctly on the TEE. Afterwards, the signature σ is verified using the master public key mpk of the TEE. Naturally, the users check that the public key stored on the contract is the same public key as the one that the enclave has created for the program p_{CT}.

If all verification steps were successful, the user proceeds to the deposit phase. Otherwise, the user aborts and does not participate in the Plasma protocol.

Deposit Phase In order to deposit coins a user P_i with public key pk_i sends a transaction to the Plasma smart contract (let v_i be the amount of coins sent by this transaction). The contract adds the tuple $(\mathsf{deposit}, pk_i, v_i)$ to a list $\mathcal{D}_\Gamma$, called *deposit list*, which stores all deposits made in the current epoch.

At the end of the deposit phase the operator sends the newly confirmed blocks to the enclave. The enclave processes the deposits, signs the deposit list $\mathcal{D}_\Gamma$ and

outputs both, the list and its signature. The operator forwards the signed list to all parties who deposited coins in the current epoch. If a user who made a deposit does not receive a valid tuple at the end of the deposit phase, she will exit in the following exit phase.

Transaction Phase In order to transfer coins, a user P_i only needs to submit a transaction $tx = (pk_i, pk_j, v, epoch_{P_i})$ with a signature σ_{pk_i} for tx under pk_i, off-chain to the operator, where pk_i, pk_j are the public keys of the sender and receiver, respectively, v is the transaction's value and $epoch_{P_i}$ is the current epoch counter. The operator collects all received transactions during this phase and then executes the transaction processing function on the enclave by giving the set of received transactions as input (see the enclave program 5.3 for more details). At the end of the transaction phase the enclave processes all transactions and returns a list of signed balances v_E, where $v_E[i]$ is the new, signed balance of user P_i. The operator sends $v_E[i] = ((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ to user P_i where σ_i is a signature for the tuple $(v_i, epoch_{\mathsf{E}}, pk_i)$ under the TEE's public key pk_{E}. If a user does not receive a correct tuple from the operator, she will exit in the next exit phase.

Exit Phase In order to withdraw money from the system a user P_i sends the signed balance value $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ that she received at the end of the transaction phase to Γ. The contract verifies that the received values are valid and that the user did not exit before, i.e., $pk_i \notin e_\Gamma$, where e_Γ is the *exit list*, which stores the public keys of all users who exited in the last two epochs. If the verification was successful the contract stores the exit in e_Γ. At the end of the exit phase the contract returns v_i coins to the exiting user P_i.

Exit Challenge If the user did not receive a valid tuple of the form $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ at the end of the transaction phase, she has to challenge the operator by sending the message $(\mathsf{exit_challenge}, pk_i)$ to the contract. The contract stores this message which indicates that the operator has been challenged and adds the user to the list of challenging parties c_Γ. In order to respond to the challenge, the operator sends the balance value $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ to the contract. This tuple is in fact the message which a user sends when she wishes to exit the system. Therefore, upon receiving this response from the operator, the contract processes this message as a regular exit made by P_i and removes the challenge from the list c_Γ.

However, if the operator does not post a valid response from the TEE until the end of the exit phase, the contract deems the operator malicious and halts the system[8]. Parties can then only send exit based on messages of the previous epoch. In a bit more detail, the contract reverts its state to the previous epoch by decrementing the epoch-counter and it announces a message indicating that the operator is malicious. The contract also returns all deposited values from the latest epoch stored in the deposit list $\mathcal{D}_\Gamma$ back to the depositing parties.

[8] Note that the operator needs to be given enough time to reply to all challenges. In practice this can be achieved by dividing the exit phase into two subphases, a challenge and a response phase.

Simplifying Assumptions In our protocol description we assumed that the contract can actively check if a phase has ended or not (which is done by checking the number of blocks produced by the blockchain). Yet in Ethereum a contract must be manually called for it to get activated and be able to check the time. Naturally, the contract checks whether or not a phase has ended whenever it receives a message but one must also add a function which can be called by any party and which checks the current time and determines if the current phase or epoch has ended.

In our protocol when a user challenges the operator in case of data unavailability, she will exit the system even if the operator responds to the challenge correctly. We note that the valid response to a challenge is in fact the signed balance which the user did not receive from $\mathcal{O}$. Hence, we can extend our protocol in order to allow users to stay in the system if $\mathcal{O}$ responds to the challenge correctly, i.e., a user P_i can indicate in her challenge message whether or not she wishes to exit upon $\mathcal{O}$ publishing $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$.

Protocol Pseudo-Code We now present the protocol program pseudo-code in Figure 2 according ot the explanation in this section.

We note that in order to prevent a user who exited in epoch $epoch_{\Gamma} - 1$ from exiting again if the operator is deemed malicious in epoch $epoch_{\Gamma}$ (as we just explained) the contract must store the list of users who exited in the previous epoch and stop such exits. Yet after epoch $epoch_{\Gamma}$ is successfully concluded, the users who exited in epoch $epoch_{\Gamma} - 1$ can be removed from the exiting list since they cannot exit again by submitting their balance from $epoch_{\Gamma} - 1$. This reduces the size of the list which is stored on the contract and allows such users to later rejoin the system by making a deposit. For simplicity, we did not mention this in the protocol code.

5.3 Enclave Program

We now describe the enclave program p_{CT} executed on the TEE. The enclave pseudo-code can be found in Appendix A in Figure 3. As mentioned above, the TEE is initialized with a master key pair (msk, mpk). To install the program on the TEE and to initialize the enclave, the operator provides the contract address Γ, the security parameter κ and the checkpoint of the ledger b_{cp} as parameters. In the following, we give an overview of the functions in the enclave program.

Generating Keys This procedure is used for the initialization of the enclave. It generates the program-specific key pair $(sk_{\mathsf{E}}, pk_{\mathsf{E}})$ for the TEE which the enclave uses to authenticate all messages with regard to p_{CT}. The key pair is also signed with the master secret key msk, which allows parties to verify it under the master public key mpk. Additionally, all internal variables, such as the list of deposits D, the list of balances v, and the list of signed balances v_E are initialized. We note that the program does not allow to execute keyGen again after this point in order to prevent the operator from resetting the system. The function returns the tuple $(pk_{\mathsf{E}}, \sigma_E)$ where σ_E is a signature for pk_{E} under mpk.

CommiTEE Protocols

Enclave Verification Protocol

P_i verifying the Enclave

1. Send the message (Verify_enclave) to $\mathcal{O}$

$\mathcal{O}$ upon (Verify_enclave) from P_i

2. Send the message $(mpk, pk_E, \sigma, \varrho)$ to P_i

P_i upon $(mpk, pk_E, \sigma, \varrho)$ from $\mathcal{O}$

3. Abort **if** $\mathsf{VrfyQuote}(mpk, p_{CT}(\kappa), (pk_E, \sigma), \varrho) \neq 1$ **or** $\mathsf{Vrfy}(mpk, pk_E, \sigma) \neq 1$ **or** $\Gamma.pk_{TEE} \neq pk_E$

Deposit Phase Protocol

P_i depositing v_i coins

1. Send $(\mathsf{deposit}, pk_i, v_i)$ to Γ

$\mathcal{O}$ at the end of deposit phase

2. Let $\bar{\mathbf{b}} := \mathbf{b}.\mathsf{get}(t + k)$
3. $(\mathcal{D}_\Gamma, \sigma_E) \leftarrow \mathsf{E}.\textsc{deposit}(\bar{\mathbf{b}})$
4. Send $(\mathcal{D}_\Gamma, \sigma_E)$ to all parties enlisted in $\mathcal{D}_\Gamma$

P_i at the end of the deposit phase

5. Exit during the exit phase **If** $(\mathcal{D}_\Gamma, \sigma_E)$ is not received **or** $\mathsf{Vrfy}(pk_E, \mathcal{D}_\Gamma, \sigma_E) \neq 1$

Γ upon $(\mathsf{deposit}, pk_i, v_i)$ from P_i

If $pk_i \notin e_\Gamma$ then add $\mathcal{D}_\Gamma := \mathcal{D}_\Gamma \cup \{(\mathsf{deposit}, pk_i, v_i)\}$

Transaction Phase Protocol

P_i sending v coins to P_j

1. Let $tx := (pk_i, pk_j, v, epoch_{P_i})$ and sign $\sigma_{pk_i} \leftarrow \mathsf{Sign}(sk_i, tx)$
2. Send $(\mathsf{transaction}, tx, \sigma_{pk_i})$ to $\mathcal{O}$

$\mathcal{O}$ upon $(\mathsf{transaction}, tx, \sigma_i)$ from P_i

3. Store $(\mathsf{transaction}, tx, \sigma_i)$ in the list of transactions made in this epoch, denoted as $\mathcal{T}_\mathcal{O}$

$\mathcal{O}$ at the end of transaction phase

4. Execute the process transactions function on the enclave and let $(v_E) \leftarrow \mathsf{E}.\textsc{process_tx}(\mathcal{T}_\mathcal{O})$
5. Send $(v_E[i])$ to each $P_i \in P$

P_i at the end of transaction phase

6. **If** $((v_i, epoch_E, pk_i), \sigma_i)$ is not received **or** $epoch_E \neq epoch_{P_i} + 1$ **or** $\mathsf{Vrfy}(pk_E, (v_i, epoch_E, pk_i), \sigma_i) \neq 1$ **then** execute the challenge procedure during the exit phase
else store the tuple $((v_i, epoch_E, pk_i), \sigma_i)$ and set $epoch_{P_i} := epoch_E$

Γ at the end of transaction phase

Set $epoch_\Gamma := epoch_\Gamma + 1$, $\mathcal{D}_\Gamma := \emptyset$ and announce (new_block_submitted)

Variable	Description
mpk	TEE master public key
pk_E	Enclave public key used in CommiTEE (stored on Γ)
pk_i	User P_i's public key
$\mathcal{D}_\Gamma$	List of deposits stored on Γ
c_Γ	List of challenges stored on Γ
e_Γ	List of submitted exits stored on Γ
$\mathtt{state}_\Gamma$	State of Γ (set to `malicious` if $\mathcal{O}$ misbehaves)
$\bar{\mathbf{b}}$	Subset of blocks from the blockchain $\mathbf{b}$
v_E	List of signed user's balances (generated by E)
$epoch_E$	E's epoch counter
$epoch_\Gamma$	Γ's epoch counter
$epoch_{P_i}$	P_i's epoch counter
t	Duration of a phase
k	Required number of confirmed blocks
ϱ	Quote generated by the the enclave

Exit Protocol

P_i requesting an exit

1. Send $(\mathsf{exit}, ((v_i, epoch_E, pk_i), \sigma_i))$ to Γ

$\mathcal{O}$ at the end of exit phase

2. Let $\bar{\mathbf{b}} := \mathbf{b}.\mathsf{get}(2t + k)$
3. Execute $\mathsf{E}.\textsc{exit}(\bar{\mathbf{b}})$

Γ upon $(\mathsf{exit}, ((v_i, epoch_E, pk_i), \sigma_i))$ from P_i

If $epoch_E = epoch_\Gamma$ **and** $\mathsf{Vrfy}(pk_E, (v_i, epoch_E, pk_i), \sigma_i) = 1$ **and** $pk_i \notin e_\Gamma$ **then** Set $e_\Gamma = e_\Gamma \cup \{(pk_i, v_i)\}$

Γ at the end of exit phase

If $c_\Gamma \neq \emptyset$ **then** set $\mathtt{state}_\Gamma := \mathtt{malicious}$ and $epoch_\Gamma := epoch_\Gamma - 1$, remove all the exit requests made in this phase from e_Γ, announce the message malicious and repeat the exit phase
else, if $\mathtt{state}_\Gamma = \mathtt{malicious}$ **then** send $v_i + v_i'$ via $\mathcal{L}$ to P_i for every tuple $(pk_i, v_i) \in e_\Gamma$ which was added to e_Γ in the current epoch where $(\mathsf{deposit}, pk_i, v_i') \in \mathcal{D}_\Gamma$
else If $\mathtt{state}_\Gamma \neq \mathtt{malicious}$ **then** send v_i via $\mathcal{L}$ to P_i for every tuple $(pk_i, v_i) \in e_\Gamma$ which was added to e_Γ in the current epoch

Exit Challenge Protocol

P_i requesting an exit challenge

1. Send (challenge_exit) to Γ

$\mathcal{O}$ upon $(\mathsf{exit_challenge}, pk_i)$ announced by Γ

2. Send $(\mathsf{respond}, v_E[i])$ to Γ

Γ upon $(\mathsf{respond}, v_E[i])$ from $\mathcal{O}$ during exit phase

3. Parse $v_E[i]$ as $((v_i, epoch, pk_i), \sigma_i)$.
4. **If** $epoch_E = epoch_\Gamma$ **and** $\mathsf{Vrfy}(pk_E, (v_i, epoch, pk_i), \sigma_i) = 1$ **then** $c_\Gamma := c_\Gamma \setminus \{pk_i\}$ and $e_\Gamma := e_\Gamma \cup \{(pk_i, b_i)\}$

Γ upon (challenge_exit) from P_i during the exit phase

If $\mathtt{state}_\Gamma \neq \mathtt{malicious}$ **and** $pk_i \notin c_\Gamma$ **and** $pk_i \notin e_\Gamma$ **then** $c_\Gamma := c_\Gamma \cup \{pk_i\}$ and announce $(\mathsf{exit_challenge}, pk_i)$

Fig. 2: Description of CommiTEE protocols.

Deposit In this procedure, the TEE adds the deposits made on-chain to the balances of the users on the enclave. This function receives as input a list of blocks $\mathbf{b}$, which is the chain of blocks since the last checkpoint.

The enclave verifies that (1) the chain $\mathbf{b}$ is a valid extension of the checkpoint and (2) the chain consists of $t + k$ blocks. We require $t + k$ blocks, since t is the duration of the phase and k are the necessary blocks to confirm the last block of the phase. If both conditions are met, the enclave extracts the deposit list $\mathcal{D}_\Gamma$ from the blocks of the deposit phase, i.e., the first t blocks of $\mathbf{b}$, updates the balances of the parties, and adds new parties to the system. Finally, the enclave computes a signature σ_E for $\mathcal{D}_\Gamma$ under pk_E and returns the tuple $(\mathcal{D}_\Gamma, \sigma_E)$ to $\mathcal{O}$.

Process Transactions This procedure is used to process transactions made by the users and to update the balances of the affected users. The parameter passed to this function is the list of transactions $\mathcal{T}_\mathcal{O}$.

For each transaction the enclave verifies that the transaction is of the form $(pk_i, pk_j, v, epoch_{P_i}, \sigma_{pk_i})$ and that σ_{pk_i} is a valid signature under pk_i. In addition, $epoch_{P_i}$ must be the current epoch and the balance of the sender must be greater or equal to v. If all these conditions are satisfied the balances of the sender and the receiver are updated according to the transaction amount.

At this point the latest balance of the users in this epoch can be updated and the epoch must be *finalized*. To this end, the TEE creates and returns the list v_E which consists per user of a tuple of the form $((v_i, epoch_\mathsf{E}, pk_i), \sigma_i)$, where σ_i is a signature under the public key of the enclave and $epoch_\mathsf{E}$ is the enclave's epoch counter.

Exit The exit procedure is used to set the balance of the users who exited to 0. The operator provides a chain of blocks $\mathbf{b}$. If the chain is valid, extends the last checkpoint and has $2t + k$ blocks (the last checkpoint was made at the end of the deposit phase, therefore at this point both transaction and exit phase are finished and hence $2t + k$ blocks must be sent to the enclave), the enclave extracts the exit list from the $2t$-th block, removes all exiting parties from the system and sets their balance to 0.

5.4 CommiTEE Security Analysis

Due to the limited space, we present the formal security properties and prove that CommiTEE is secure in Appendix C. Here, we briefly discuss why our protocol achieves the security properties from Section 4.

Theorem 1 (informal). *The* CommiTEE *protocol as described in Section 5 satisfies the correctness, security and efficiency properties as described in Section 4.*

The most challenging property to prove is security as it involves a malicious operator who can behave arbitrarily. We first shortly discuss why correctness and efficiency are satisfied.

It is easy to see that CommiTEE satisfies correctness, since in case the operator is honest, she will honestly provide the list of deposits, transactions and exits to her

TEE which honestly updates and returns the signed balances of the users. All users receive their respective signed balances since the operator is honest. Hence, deposit, transaction and exit phase correctness is satisfied. Further, as each phase of the protocol has a constant duration, the efficiency property is satisfied.

Let us now discuss the three security properties. Due to the usage of a TEE which acts as a trusted entity, a malicious operator can only mount data unavailability attacks, i.e., refuse to forward data between users and the TEE. Data unavailability attacks are a general issue in Plasma protocols and can never be prevented. However, we show that CommiTEE provides sufficient mechanisms to protect honest parties in this case. Assume an operator who does not send the signed balances from the TEE to users after a transaction phase. In this case, the users will challenge the operator on-chain and exit the system. If the operator responds to the challenge with a signed balance tuple generated by the TEE, the user exits based on her latest balance. Otherwise, the contract announces the operator as malicious and all users exit based on their balance from the previous epoch. In this case, the contract returns all coins which were deposited in this epoch. Note that as commit-chain protocols satisfy late finality, users should be able to exit either with their balance from the previous or current epoch. Hence, balance and deposit security are satisfied. Finally, as our protocol does not require the operator to lock any collateral, operator balance security is satisfied.

6 Evaluation

We evaluated CommiTEE's costs both in terms of gas costs and on-chain communication complexity. In order to evaluate the gas costs we used Ganache-cli [18] to simulate full client behaviour. The contract itself is written for Solidity version 0.5.3. Our implementation can be found at [14]. To evaluate the communication complexity we analyzed the size (in bytes) of the parameters sent for each function call of the contract. In this evaluation (Table 2), we did not include the size overhead of sending a transaction, i.e., for function calls without any parameters we assume a size of 0 bytes.

We compare the results of our evaluation with the most widely known commit-chain protocol namely NOCUST and NOCUST-ZKP [24] and with the most common Plasma protocols, namely Plasma MVP and Plasma Cash.[9] We use the evaluation results from [24] in order to compare CommiTEE with NOCUST/NOCUST-ZKP in terms of gas costs. On the other hand, most implementations of Plasma MVP and Plasma Cash are experimental and neither optimized in terms of gas costs nor do they execute flawlessly without errors. Therefore our comparison with Plasma Cash and MVP is with respect to the message size of all (potential) interaction that occurs during the protocol execution. Note that our protocol is fundamentally different to both Plasma MVP and Plasma Cash. In Plasma MVP and Plasma Cash, a malicious user can attempt to exit another user's coins by sending an exit request of an already spent UTXO or coin respectively. Hence, users must constantly observe

[9] For comparison we used the omiseGO and loom network respectively [36, 29].

the blockchain and challenge malicious behavior on-chain which might be expensive or problematic in case of blockchain congestion. On the other hand, in our protocol users only challenge the operator in case of data unavailability (which is unavoidable according to the work of Dziembowski et.al., [16]), i.e., in case the operator does not provide the signed balances to all users. We would like to point out that there are many different proposals (other than Plasma MVP and Cash) on how to design a Plasma protocol such as Plasma Snapp [39], Plasma Debit [38], and More Viable Plasma [34]. However, these are only proposals mentioned on online forums and are not formally specified. Therefore, we do not compare our protocol with these approaches.

6.1 Comparison with NOCUST(-ZKP)

Let us first compare CommiTEE with the best known commit-chain protocol namely NOCUST and NOCUST-ZKP [24]. The main difference between NOCUST and NOCUST-ZKP is that NOCUST-ZKP utilizes zero knowledge proofs in order to guarantee valid state transitions where NOCUST allows users to challenge the operator in case the state transaction is invalid. Therefore, NOCUST-ZKP is from a design perspective closer to CommiTEE. However, since CommiTEE uses a TEE there is no need to submit and verify expensive zero knowledge proofs on-chain in order to guarantee that the state transition is valid. The full comparison can be found in Table 1. As we can see, CommiTEE is almost 3 times cheaper when finalizing an epoch compared to NOCUST and more than 19 times cheaper than NOCUST-ZKP. We would like to point out that our evaluated gas cost does not increase with the number of users n or transactions v. Furthermore, the reported gas cost for NOCUST is with respect to a system with only 10 users who only make 20 transaction, i.e., $n = 10$ and $v = 20$.

Function	Gas Cost		Paid By	Complexity	
	CommiTEE	NOCUST(-ZKP)		CommiTEE	NOCUST(-ZKP)
Deposit	69 815	64 720	User	$O(1)$	$O(1)$
Exit	118 601	169 238	User	$O(1)$	$O(\log(n))$
Exit Challenge	66 548	225 642	User	$O(1)$	$O(\log(n) + \log(v))$
Exit Response	74 580	68 152	Operator	$O(1)$	$O(\log(n) + \log(v))$
Total Exit Challenge/Response	141 128	293 794	User and Operator	$O(1)$	$O(\log(n) + \log(v))$
Finalization	32 363	96 073	Operator	$O(1)$	$O(1)$
ZK-proof	-	>523 618	Operator	-	$O(1)$
Total Finalization cost	32 363	>619 691	Operator	$O(1)$	$O(1)$
State Challenge	-	281 786	User	-	$O(\log(n))$
State Response	-	80 769	Operator	-	$O(\log(n))$

Table 1: Comparison of CommiTEE with NOCUST and NOCUST-ZKP with regard to gas cost and on-chain communication complexity. The gas cost in row "ZK-proof" is only relevant for NOCUST-ZKP, while the gas costs in rows "State Challenge" and "State Response" are only relevant for NOCUST. The evaluated gas cost for NOCUST(-ZKP) are for $n = 10$ and $v = 20$ where the parameters n and v represent the number of users and transactions respectively.

6.2 Comparison with Plasma MVP and Cash

Deposit For depositing in Plasma MVP and CommiTEE, a party only has to send a transaction with the amount of coins that it wants to deposit into the contract, which then can extract and store the sender identifier and the transaction value. On the other hand, when depositing in Plasma Cash, a user must send some additional information such as a coin identifier and a user address to the contract, resulting in an overhead of 105 bytes.

Exit and Finalize Exits In order to implement the exit mechanism, Plasma contracts require two functions, namely an initiating and finalizing Exit function.

1. **Initiating Exit.** In order to initiate an exit, a user first has to send an exit request to the Plasma contract. In CommiTEE the user sends the signed balance value to the contract to request an exit. In contrast, in Plasma MVP to initialize an exit the user needs to send the position of the UTXO, the UTXO itself, the signature of the UTXO, a signature confirming the inclusion of the UTXO and a Merkle proof for this inclusion. In Plasma Cash the exit request consists of the token to be exited, two Merkle proofs (for the current owner of the token and the previous owner of the token [17]) together with two signatures and the position in the Plasma chain, i.e., the epoch counter. This exit request is stored in the contract and can be challenged while the exit phase is running.
2. **Finalizing Exit.** At the end of the exit phase the contract is called again, in order to process all stored exit requests. In CommiTEE the exit finalization does not require any additional information, however Plasma MVP and Cash require an additional list which indicates the exit requests that should be finalized in this epoch. This is because in Plasma MVP and Cash an exit request does not need to be processed in the same epoch. In fact, the challenge period is usually set to 7 days [9] in order to give honest users enough time to challenge malicious exit requests.

Exit Challenge and Response As mentioned before, all three Plasma systems allow parties to submit a challenge in case they suspect malicious behavior. We emphasize that there is a fundamental difference between the challenges in CommiTEE and Plasma MVP or Cash. In CommiTEE the users challenge the operator in case of data unavailability, in order to exit the system. In comparison, in Plasma MVP or Plasma Cash users issue a challenge when a malicious user attempts to steal their coins. Furthermore, in both Plasma MVP and Plasma Cash parties need to exit in case of data unavailability, where in Plasma MVP this must be done immediately while in Plasma Cash users do not need to rush and it suffices to eventually exit. Hence, in practice our protocol does not require users to monitor other user's exits in order to save their coins, which is a significant improvement over the other variants of Plasma.

Finalize Finally, in both Plasma MVP and Cash the operator must submit additional information such as the Merkle root (of the Merkle tree which commits to

the transactions or coins) on the ledger. In contrast, in CommiTEE the operator does not have to submit such a message to the contract, thus saving additionally on communication with the ledger.

Overall, our protocol substantially reduces the communication complexity with the ledger. In order to evaluate our implementation, we estimated the gas costs for deposits, exits in the honest and malicious cases and for the finalization. We also analyzed the size of the parameters that are needed to call the different functions on the contract and compared them to other implementations (Plasma MVP and Plasma Cash). The overview of our results can be found in tables 1 and 2.

Table 2 shows the comparison of the message size in bytes for each of the protocols CommiTEE, Plasma MVP and Plasma Cash and for each respective phase of an epoch.

Function	CommiTEE	MVP	Cash
Deposit	0	0	105
Exit	117	≥ 266	$\geq 449 \cdot n$
Exit Challenge	20	≥ 323	≥ 253
Exit Response	117	-	≥ 285
Epoch Finalize	0	32	64

Table 2: Sizes in bytes. Only counting function parameters, while abstracting from constant transaction size. n represents the balance of a user in Plasma Cash.

7 Extensions to CommiTEE

In this section, we discuss two extensions to CommiTEE , namely how to support multiple operators and how to deal with TEE compromise. Both extensions significantly increase security and applicability of CommiTEEas users can continue using the system even in presence of a malicious operator or a compromised TEE.

7.1 Supporting Multiple Operators

Most previous works on commit-chain or Plasma protocols [9, 10, 24] assume that only one operator maintains the system as multi-operator support would either require to establish consensus among all operators on the latest state of the system, or it would require users to publish the latest state of the system on-chain. The first approach introduces huge communication overhead on the operators and also requires an honest majority assumption among the operators. The second approach, however, is not much different to requiring all users to exit the protocol and deposit their coins into a new Plasma system.

By leveraging a TEE, CommiTEE can support multiple operators and avoid the above mentioned challenges. In our solution the backup operators (i.e., all operators except for the currently active one) remain idle until the active operator is deemed malicious by the contract Γ.

We now elaborate on how to extend our system to a multi-operator setting. For simplicity, we consider the setting of two operators, as it is straightforward to extend our approach to more than two operators. Assume $\mathcal{O}_1$ is the active operator, while $\mathcal{O}_2$ is a passive backup operator. Upon setup of the contract Γ, $\mathcal{O}_1$ and $\mathcal{O}_2$ register their public keys in a list L in the contract, such that $\mathcal{O}_1$'s public key is the first element in L. $\mathcal{O}_1$ acts as described in our protocol description from Section 5; $\mathcal{O}_2$ on the other hand only needs to monitor Γ every epoch to check if Γ marks $\mathcal{O}_1$ as malicious. As long as $\mathcal{O}_1$ is not marked as malicious, $\mathcal{O}_2$ can stay inactive.[10]

However, when Γ announces $\mathcal{O}_1$ as malicious, it extracts the public key of $\mathcal{O}_2$ from L (i.e., the next element in L) and registers $\mathcal{O}_2$ as the next active operator. The contract additionally removes the public key of $\mathcal{O}_1$ from L. Upon being announced as the new active operator, $\mathcal{O}_2$ first has to send a confirmation message to Γ within a pre-defined time period Δ. If Γ does not receive this confirmation from $\mathcal{O}_2$ within Δ time, then all users have to exit the system as described in the exit protocol of CommiTEE by submitting the message $(\mathsf{exit}, (v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i))$ to Γ.[11]

Otherwise, if $\mathcal{O}_2$ confirms the operator switch, users have two options, namely either to exit or stay in the system. In the latter case, users first verify that the enclave in $\mathcal{O}_2$'s TEE has been initialized correctly (i.e., that the correct program is installed on the TEE of $\mathcal{O}_2$), as described in Section 5.2. Upon successful verification, each user P_i signs and sends the message $((\mathsf{swap}_{\mathcal{O}}, (v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i), \sigma_{pk_i})$ to $\mathcal{O}_2$, where the tuple $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ represents the user's balance in epoch $epoch_{\mathsf{E}}$ and σ_{pk_i} is a signature under P_i's public key pk_i. Note that this tuple also contains a valid signature σ_i of $\mathcal{O}_1$'s TEE which serves as proof for P_i's balance in epoch $epoch_{\mathsf{E}}$.

Upon receiving these messages, $\mathcal{O}_2$ (by using its TEE) checks if the signatures σ_i and σ_{pk_i} are valid under the public key of $\mathcal{O}_1$'s TEE and pk_i respectively. If so, the TEE stores the balance of this user and outputs a message $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i')$ to P_i, where σ_i' is a valid signature with respect to the public key of $\mathcal{O}_2$'s TEE. Naturally, if the user does not receive this message from $\mathcal{O}_2$ (i.e., $\mathcal{O}_2$ is also malicious), she submits the exit message $(\mathsf{exit}, (v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ to Γ.

At the end of this epoch, $\mathcal{O}_2$ forwards the list of parties who exited on Γ to its TEE. The TEE checks if any of the users who agreed to swap the operator have exited and if so deletes their information. Note that a party cannot exit twice (by submitting both $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ and $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i'))$ because Γ does not allow the same pk_i to exit twice.

7.2 Handling TEE Compromise

Naturally the security of our protocol relies on the security of the TEE. However, by making some modifications, it is possible to detect if the TEE was compromised in certain situations.

[10] We emphasize that $\mathcal{O}_2$ does not need to interact with $\mathcal{O}_1$ at any stage of the protocol.

[11] Note that if there was a third operator in L, the contract would register her as the next active operator.

We consider a situation where the party's expected final balance is different from what she has received from the operator. In this case either the operator did not forward some transactions to the TEE or the TEE has been compromised.

This extension requires that a transaction in CommiTEE is signed by both, the sender and the receiver. Under this requirement an honest user P_i can compute her final balance $v_{i,e}$ after the transaction phase of an epoch e, since she knows the list of transactions $TX_{i,e}$ consisting of all transactions that she sent and received in epoch e and she knows her starting balance $v_{i,e-1}$ from the previous epoch. As such, if P_i receives a final balance $\tilde{v}_{i,e} \neq v_{i,e}$ from the operator after the transaction phase, she can draw one of the following conclusions: (1) the operator did not forward some transactions in $TX_{i,e}$ to the TEE or (2) the integrity of the TEE has been compromised[12]. As mentioned before, the former case is a form of data unavailability attack from the operator, which cannot be prevented. However, P_i can distinguish cases (1) and (2), as she is able to compute all possible combinations of transactions in $TX_{i,e}$ and check if any of these combinations results in the balance $\tilde{v}_{i,e}$. In case she finds such a combination, she concludes that case (1) happened (notice that even if this was a result of a compromised TEE, the effect is the same as in case of a data unavailability attack). Otherwise, P_i concludes that the TEE must be corrupted.[13] In this case, the user can challenge the operator on-chain, which requires the operator to publish the set of transactions which results in the final balance $\tilde{v}_{i,e}$ as output by the TEE. If the operator cannot do so, the contract Γ announces that the TEE has been compromised and users can switch to another operator as described in Section 7.1. If the operator can answer the challenge correctly, then the user maliciously challenged the operator.

While this solution works well for applications with moderate transaction rate per epoch, it does not scale to use cases with high transaction frequency per epoch as the computation of all possible final balances grows significantly with large transaction sets.

7.3 Conclusion

In this work we have designed CommiTEE, an efficient and secure Plasma protocol which requires minimum on-chain interaction. By using zero-knowledge proofs, the on-chain cost of the most prominent existing Plasma solution NOCUST-ZKP increases by a factor of 19 when compared to our protocol. Furthermore, compared to other well-known protocols such as Plasma MVP or Cash our protocol reduces the on-chain communication complexity by at least 2 times (and in some cases more than 16 times). As an additional contribution, we present the first model for Plasma/Commit-Chain protocols, which paves the way for rigorous security analyses of existing and future Plasma protocols.

Finally, we have shown how to extend CommiTEE in order to incorporate multiple operators in the system, which allows users to switch from a malicious operator to an honest one. Our approach does not require the operators to run a consensus

[12] We assume that all transactions in $TX_{i,e}$ are valid.
[13] This follows from the fact that the operator cannot forge signatures under the P_i's public key.

mechanism, in fact the backup operators do not need to communicate with the active operator at all. This extension improves the usability of CommiTEE in case the active operator acts maliciously, crashes or loses connection.

We are convinced that our results will not only affect the huge landscape of Plasma (Commit-Chain) protocols but also proves that it is indeed possible to design efficient and practical Commit-Chain protocols.

There are multiple directions in which our work can be extended. As the technology of TEEs gets more and more mature and ready for widespread use, it might be interesting to consider a Plasma protocol where not only the operators but also all users operate a TEE. This might be a great way to reduce blockchain interaction even further or to provide even stronger security guarantees. Furthermore, considering a Plasma/Commit-Chain protocol that supports executing smart contracts off-chain is an interesting direction for future work.

References

[1] M. Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 1 (2009).

[2] H. Amler et al. "DeFi-ning DeFi: Challenges & Pathway". In: *CoRR* (2021). arXiv: `2101.05589`.

[3] I. Bentov et al. "Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* London, United Kingdom, 2019.

[4] A. Biondo et al. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel {SGX}". In: *27th {USENIX} Security Symposium ({USENIX} Security 18).* 2018.

[5] *Bitcoin Wiki: Payment Channels.* `https://tinyurl.com/y6msnk7u`.

[6] M. Bowman et al. "Private Data Objects: an Overview". In: *ArXiv* (2018).

[7] F. Brasser et al. "Software grand exposure:{SGX} cache attacks are practical". In: *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17).* 2017.

[8] V. Buterin. *Endgame.* `https://vitalik.ca/general/2021/12/06/endgame.html`. 2021.

[9] V. Buterin. *Minimal Viable Plasma.* `https://tinyurl.com/y2s9grpd`. 2018.

[10] V. Buterin. *Plasma Cash.* `https://tinyurl.com/y5fjm2t9`. 2018.

[11] V. Buterin. *The Dawn of Hybrid Layer 2 Protocols.* `https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html`. 2019.

[12] *ChainLink.* `https://chain.link/`.

[13] R. Cheng et al. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *2019 IEEE European Symposium on Security and Privacy* (2019).

[14] *CommiTEE.* `https://tinyurl.com/y4yb7xv2`.

[15] P. Das et al. *FastKitten: Practical Smart Contracts on Bitcoin*. Cryptology ePrint Archive, Report 2019/154. https://eprint.iacr.org/2019/154. 2019.
[16] S. Dziembowski et al. "Lower Bounds for Off-Chain Protocols: Exploring the Limits of Plasma". In: ().
[17] K. Floersch. *Plasma Cash Simple Spec*. https://tinyurl.com/yxdp2rqr. 2018.
[18] *Ganache-cli*. https://github.com/trufflesuite/ganache-cli.
[19] M. Harishankar et al. *PayPlace: A Scalable Sidechain Protocol for Flexible Payment Mechanisms in Blockchain-based Marketplaces*. 2020. arXiv: 2003.06197 [cs.CR].
[20] M. Hoekstra et al. "Using Innovative Instructions to Create Trustworthy Software Solutions". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel, 2013.
[21] *Intel. 2021. 3rd Gen Intel Xeon Scalable Processors Brief*. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html.
[22] A. Juels et al. "The Ring of Gyges: Investigating the Future of Criminal Smart Contracts". In: 2016.
[23] G. Kaptchuk et al. "Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*.
[24] R. Khalil et al. "Commit-Chains: Secure, Scalable Off-Chain Payments". In: ().
[25] A. Kosba et al. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016.
[26] L. Lamport et al. "The Byzantine generals problem". In: *Concurrency: the Works of Leslie Lamport*. 2019.
[27] A. Limited. *Building a Secure System using TrustZone Technology*. https://tinyurl.com/amhjkfxy. 2009.
[28] J. Lind et al. "Teechain: A Secure Payment Network with Asynchronous Blockchain Access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville, Ontario, Canada, 2019.
[29] *loom network*. https://github.com/loomnetwork/plasma-cash.
[30] G. Malavolta et al. "Concurrency and Privacy with Payment-Channel Networks". In: 2017.
[31] S. Matetic et al. "BITE: Bitcoin Lightweight Client Privacy using Trusted Execution". In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019.

[32] F. McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Tel-Aviv, Israel, 2013.
[33] R. Mitra. *Plasma Breakthrough: OmiseGO (OMG) announces the launch of Ari*. `https://tinyurl.com/y262ttos`. (Accessed on 02/08/2020).
[34] *More Viable Plasma*. `https://tinyurl.com/y2gpmwov`. 2018.
[35] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).
[36] *omiseGO*. `https://github.com/omgnetwork/plasma-mvp`.
[37] R. Pass et al. "Formal abstractions for attested execution secure processors". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017.
[38] *Plasma Debit*. `https://tinyurl.com/yx936xzk`. 2018.
[39] *Plasma snapp*. `https://tinyurl.com/yxbza3pl`. 2018.
[40] J. Poon and V. Buterin. "Plasma: Scalable autonomous smart contracts". In: (2017).
[41] J. Poon and T. Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. `https://tinyurl.com/q54gnb4`. 2016.
[42] K. Qin et al. "CeFi vs. DeFi–Comparing Centralized to Decentralized Finance". In: *arXiv preprint arXiv:2106.08157* (2021).
[43] M.-W. Shih et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In: *NDSS*. 2017.
[44] Trustnodes. *Ethereum Transactions Fall Off the Cliff, Three Plasma Projects Close to Release Says Buterin*. 2018.
[45] J. Van Bulck et al. "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018.
[46] S. M. Werner et al. "SoK: Decentralized Finance (DeFi)". In: *CoRR* (2021). arXiv: `2101.08778`.
[47] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* (2014).
[48] K. Wüst et al. "ZLiTE: Lightweight Clients for Shielded Zcash Transactions Using Trusted Execution". In: *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. 2019.
[49] F. Zhang et al. "Paralysis Proofs: Secure Dynamic Access Structures for Cryptocurrency Custody and More". In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. Zurich, Switzerland, 2019.
[50] F. Zhang et al. "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria, 2016.
[51] *ZK Rollups*. `https://tinyurl.com/y4rxp4az`. 2020.

A Enclave Code

In this section, we present the enclave program pseudo-code corresponding to the explanation in section 5.3. The protocol pseudo-code was already provided in Figure 2 and the enclave pseudo-code can be found in Figure 3.

B Formal Properties

B.1 Formal Properties

In this section we formally define the properties that a Plasma protocol must satisfy.

First let us define some notation. We denote the set of honest parties as $\mathcal{H} \subseteq \mathcal{P} \cup \{\mathcal{O}\}$, the set of all finished epochs is referred to as $\mathcal{E}$ and a single epoch from the set in the set of finished epochs is denoted as $\varepsilon \in \mathcal{E}$. The formal definition of the properties depend on the input and output of the users during each phase of the epochs. Therefore, we use the following notation for a protocol execution. Note that for the protocol description we omitted mentioning these input output behavior (which is only required in our security analysis) for clarity and in order to be concise.

Protocol execution

A protocol is executed between the users, the operator, and the contract. Therefore, we consider every protocol as a $n+2$ party protocol π where n denotes the total number of users. For every protocol we first define an input domain $\mathcal{D}_{\mathit{in}\text{-}\pi}$. This domain specifies which values may be used as inputs to the protocol. Similarly we use an output domain $\mathcal{D}_{\mathit{out}\text{-}\pi}$ which specifies the possible outputs for a protocol execution.

In order to model the presence of an adversary $\mathcal{A}$ who can corrupt parties, we introduce the following notation for a protocol execution during an epoch ε similar to the notation used in [15].

$$(\mathbf{O}, \mathbf{acc}^{new}, \mathcal{P}^{new}) \leftarrow REAL_{\pi,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{old})$$

$\mathbf{I} \in \mathcal{D}^{n}_{\mathit{in}\text{-}\pi}$ is the input vector and $\mathbf{I}[i]$ is the input of $P_i \in \mathcal{P}$. $\mathbf{O} \in \mathcal{D}^{n+2}_{\mathit{out}\text{-}\pi}$ is defined analogously where $\mathbf{O}[\mathcal{O}]$ and $\mathbf{O}[\Gamma]$ defines the output of the operator and contract respectively. Note that parties may have a set of inputs or outputs (e.g., a party might make multiple transactions in an epoch). $\mathbf{acc}^{new}, \mathbf{acc}^{old}$ denote the balance vectors before and after the protocol execution. Concretely $\mathbf{acc}^{old}[i]$ defines the balance of P_i before the protocol execution. The values $\mathcal{P}^{old}$ and $\mathcal{P}^{new}$ denote the set of parties in the Plasma network before and after the protocol execution.

Let us now define the input and the output behavior for the deposit, transaction, and exit phases.

CommiTEE 's Enclave Code

Variable	Description
msk, mpk	TEE master secret and public keys
sk_E, pk_E	Enclave secret and public keys for CommiTEE
s	Name of the next function that can be called
v	List of user's balances
v_E	List of signed user's balances
P	Set of users
$epoch_E$	E's Epoch counter
b_{cp}	Blockchain checkpoint stored on E

Algorithm 1 Enclave Program p_{CT} Key Generation, Deposit Transaction and Exit Phases

```
 1: s = keyGen
 2: procedure GENKEYS(κ)
 3:     if s = keyGen then
 4:         (sk_E, pk_E) ← Gen(1^κ)
 5:         σ_E ← Sign(msk; pk_E)
 6:         v := []
 7:         v_E := []
 8:         P := ∅
 9:         epoch_E := 0
10:         s := deposit
11:         return (pk_E, σ_E)
12:     end if
13: end procedure
14: procedure DEPOSIT(b̄ = (b_l, ..., b_{l+t}, ..., b_{l+t+k}))
15:     if VrfyChain(b_cp, b̄) = 1 and s = deposit then
16:         Extract the deposit list D_Γ from b_{l+t}
17:         for each d ∈ D_Γ do
18:             parse d as (deposit, pk_i, v_i)
19:             P := P ∪ {pk_i}
20:             v[i]= v[i] + v_i
21:         end for
22:         σ_E ← Sign(sk_E, D_Γ)
23:         b_cp := b_{l+t+k}
24:         s := transaction
25:         return (D_Γ, σ_E)
26:     end if
27: end procedure
28: procedure PROCESS_TX(T_O)
29:     if s = transaction then
30:         for each tx in T_O do
31:             Parse tx as (pk_i, pk_j, v, epoch_{P_i}, σ_{pk_i})
32:             if Vrfy(pk_i, (pk_i, pk_j, v, epoch_{P_i}), σ_{pk_i}) = 1 and epoch_{P_i} + 1 = epoch_E and v ≤ v[i] then
33:                 v[i] := v[i] − v
34:                 v[j] := v[j] + v
35:             end if
36:         end for
37:         epoch_E := epoch_E + 1
38:         v_E := ∅
39:         D := ∅
40:         for each P_i ∈ P do
41:             σ_i ← Sign(sk_E; (v[i], epoch_E, pk_i))
42:             v_E := v_E ∪ {((v[i], epoch_E, pk_i), σ_i)}
43:         end for
44:         s := exit
45:         return (v_E)
46:     end if
47: end procedure
48: procedure EXIT((b̄ = (b_l, ..., b_{l+2t}, ..., b_{l+2t+k})))
49:     if VrfyChain(b_cp, b̄) = 1 and s = exit then
50:         Extract the exit list e_Γ from b_{l+2t}
51:         for each (pk_i, ·) in e_Γ do
52:             v[i] := 0
53:             P := P \ {pk_i}
54:         end for
55:         b_cp := b_{l+2t+k}
56:         s := deposit
57:     end if
58: end procedure
```

Fig. 3: Pseudo-code of CommiTEE 's Enclave program.

Deposit Phase The input domain $\mathcal{D}_{\textit{in-d}} := \{\emptyset, (\mathsf{deposit}, v)\}$ denotes the input values each user may provide. $\emptyset$ indicates that a user did not deposit coins during the protocol execution. $(\mathsf{deposit}, v)$ specifies the amount v which was requested as deposit. The output domain is denoted as $\mathcal{D}_{\textit{out-d}} := \{\emptyset, (\mathsf{deposited}, P_i, v)\}$ where $(\mathsf{deposited}, P_i, v)$ indicates that the deposit of P_i with the amount v was processed. For simplicity we assume that a user makes at most one deposit during the deposit phase.

Transaction Phase The input domain for the transaction phase is defined as $\mathcal{D}_{\textit{in-t}} := \{\emptyset, \{tx\}\}$ where $tx := (pk_i, pk_j, v)$ denotes the transaction that party pk_i submitted to $\mathcal{O}$ and $\emptyset$ indicates that no transaction was submitted. The output domain is denoted as $\mathcal{D}_{\textit{out-t}} := \{\emptyset, \{tx\}\}$ where $\{tx\}$ is the set of transactions that were processed.

Exit Phase The input domain for the exit phase is defined as $\mathcal{D}_{\textit{in-e}} := \{\emptyset, (\mathsf{exit})\}$ where exit indicates that an exit was requested. The output domain is defined as follows:

$$\mathcal{D}_{\textit{out-e}} \subseteq \{(\mathsf{exited}, P_i, v), (\mathsf{deposit\text{-}returned}, P_i, v')\}$$

With the output $(\mathsf{exited}, P_i, v)$ indicating that Γ processed an exit by P_i and sent the amount v to P_i on $\mathcal{L}$ and $(\mathsf{deposit\text{-}returned}, P_i, v')$ indicating that Γ returned the deposit that was made on $\mathcal{L}$. The output domain of Γ can be a set of exited and $\mathsf{deposit\text{-}returned}$ messages.

Correctness Properties

The correctness properties describe how the balances of the users are updated. There are two correctness properties, namely (1) deposit and transaction phase correctness (2) exit phase correctness

Deposit and Transaction Phase Correctness Intuitively deposit phase correctness ensures that if an honest party deposits coins on the contract and the operator is honest, the balance and the set of users is updated accordingly. Transaction phase correctness ensures that if the sender of a transaction and the operator are honest, the transaction is included and the user's balance are updated accordingly. Since Plasma protocols achieve late finality, both these properties must hold at the end of the transaction phase. More formally we have:

For all epochs $\varepsilon \in \mathcal{E}$, input vectors $\mathbf{I}_d \in \mathcal{D}^n_{\textit{in-d}}$ and $\mathbf{I}_t \in \mathcal{D}^n_{\textit{in-t}}$ and balance vectors $\mathbf{acc}^{old}$, let the output of the deposit and transaction protocols be $(\cdot, \mathbf{acc}^{old}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_d,\mathcal{A},\varepsilon}(\mathbf{I}_d, \mathbf{acc}^{old}, \mathcal{P}^{old})$, $(\mathbf{O}, \mathbf{acc}^{new}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_t,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{new})$ respectively. Furthermore, let $\mathsf{Tx}^s_{P_i} \subseteq \mathbf{O}[\mathcal{O}]$ be the set of transaction of the form $(pk_i, \cdot, v)$ (i.e., transaction sent by P_i) and let $\mathsf{Tx}^r_{P_i} \subseteq \mathbf{O}[\mathcal{O}]$ be the set of transaction of the form $(\cdot, pk_i, v)$ (i.e., transaction received by P_i). If $P_i, \mathcal{O} \in \mathcal{H}$ the following must hold:

$$\begin{aligned} \mathbf{I}[i] =& \mathbf{O}[i] = \mathsf{Tx}^s_{P_i} \\ \mathbf{acc}^{new}[i] =& \mathbf{acc}^{old}[i] + x + \sum_{(\cdot,\cdot,v)\in \mathsf{Tx}^r_{P_i}} v - \sum_{(\cdot,\cdot,v')\in \mathsf{Tx}^s_{P_i}} v' \end{aligned}$$

where if $\mathbf{I}_d[i] = (\mathsf{deposit}, d)$ then $x = d$ and otherwise $x = 0$.

Exit Phase Correctness On a high level exit phase correctness states that if an honest party exits and the operator is also honest the balance and the user set will be updated accordingly. This means that balance of the exiting party is set to 0 and that the party is removed from the user set. More formally we have:

For all epochs $\varepsilon \in \mathcal{E}$, input vectors $\mathbf{I} \in \mathcal{D}^n_{in\text{-}e}$ and balance vectors $\mathbf{acc}^{old}$, let the protocol output be $(\mathbf{O}, \mathbf{acc}^{new}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_e,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{old})$, then $\forall P_i \in \mathcal{H} : \mathbf{I}[i] = \mathsf{exit}$ if $\mathcal{O} \in \mathcal{H}$ it must hold that:

$$\mathbf{acc}^{new}[i] = 0$$
$$\mathcal{P}^{new} = \mathcal{P}^{old} \setminus \{P_i\}$$

Security

We now describe the security properties that a plasma protocol must satisfy.

Deposit Security Intuitively, deposit security states that if the deposit of an honest user is not processed correctly, then the user receives the deposited value at the end of the exit phase.

For all epochs $\varepsilon \in \mathcal{E}$, input vectors $\mathbf{I} \in \mathcal{D}^n_{in\text{-}d}$ and balance vectors $\mathbf{acc}^{old}$, let the protocol output be $(\mathbf{O}, \mathbf{acc}^{old}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_d,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{old})$, then $\forall P_i \in \mathcal{H} : \mathbf{I}[i] = (\mathsf{deposit}, v)$ where $\mathbf{O}[i] \neq (\mathsf{deposited}, P_i, v)$, it must hold that:

$$(\mathsf{deposit\text{-}returned}, P_i, v) \in \mathbf{O}e[i]$$
$$\wedge(\mathsf{deposit\text{-}returned}, P_i, v) \in \mathbf{O}e[\Gamma]$$

where $\mathbf{O}e$ is the output of the protocol at the of the exit phase.

Balance Security On a high level Balance Security states that an honest user can always either exit her balance from the current epoch or the previous epoch. We note that if the user exits accordion to her balance from the previous epoch, she will also receive the amount of coins that she deposited in this epoch.

For all epochs $\varepsilon \in \mathcal{E}$, input vectors $\mathbf{I} \in \mathcal{D}^n_{in\text{-}e}$ and balance vectors $\mathbf{acc}^{old} \in \mathbb{N}^n$, let the exit protocol output be $(\mathbf{O}, \mathbf{acc}^{new}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_e,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{old})$, the $\forall P_i \in \mathcal{H} : \mathbf{I}[i] = \mathsf{exit}$ one of the following holds:

$$(\mathsf{exited}, P_i, \mathbf{acc}^{old}[i]) = \mathbf{O}[i] \in \mathbf{O}[\Gamma]$$

or

$$\mathsf{exited}, P_i, \mathbf{acc}^{old'}[i]) \in \mathbf{O}[i]$$
$$\wedge(\mathsf{exited}, P_i, \mathbf{acc}^{old'}[i]) \in \mathbf{O}[\Gamma]$$

where $\mathbf{acc}^{old'}[i]$ is the balance of the user at the beginning of the depost phase of epoch ε.

Operator Security Operator Security states that an honest operator does not lose the money she deposited on the contract. This implies that for any protocol where the operator has to provide a collateral an exit mechanism to withdraw that collateral has to be provided.

More formally, let ε_0 denote the first epoch and let v denote the initial balance stored on Γ in ε_0, then for all epochs $\varepsilon \in \mathcal{E}$, input vectors $\mathbf{I} \in \mathcal{D}^n_{in\text{-}e}$ and balance vector, $\mathbf{acc}^{old} \in \mathbb{N}^n$, let the exit protocol output be $(\mathbf{O}, \mathbf{acc}^{new}, \mathcal{P}^{new}) \leftarrow REAL_{\pi_e,\mathcal{A},\varepsilon}(\mathbf{I}, \mathbf{acc}^{old}, \mathcal{P}^{old})$, then if $\mathcal{O} \in \mathcal{H}$, $\mathbf{I}[\mathcal{O}] = \mathsf{exit}$ and $\mathcal{P}^{new} = \emptyset$, the following holds:

$$\mathbf{O}[\mathcal{O}] = (\mathsf{exited}, \mathcal{O}, v') \in \mathbf{O}[\Gamma]$$

Where $v' \geq v$.

Efficiency

Protocol Efficiency Let the duration of an epoch $\varepsilon \in \mathcal{E}$ be denoted as δ. A Plasma protocol is efficient if it holds that $\delta \in O(1)$ for every $\varepsilon \in \mathcal{E}$.

C Proof of Plasma Properties

C.1 Security Analysis

In this section we argue that CommiTEE as described in section 5 satisfies the properties defined in Appendix B. We show that the correctness, security, efficiency, properties are satisfied (except with negligible probability) in our model.

We analyze the relevant steps of each protocol and argue why these steps result in satisfying the required properties.

Assumptions

Let us shorty recall our assumptions and model. First we assume the adversarial model of the Plasma framework which we introduced in section 4.1. I other words we consider a byzantine adversary [26], a secure underlying ledger which can execute smart contracts and a stable network in which the parties are connected via authenticated channels.

We assume that the TEE is secure as described in section 2.2. This implies that it is infeasible for an adversary to forge a valid quote ϱ or mount any attacks which would compromise the TEE.

Furthermore, we assume that $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, the signature scheme used in the protocol, is existentially unforgeable under chosen message attack.

Correctness

In order to prove that our protocol satisfies correctness, we show that deposit and transaction phase correctness and exit phase correctness are satisfied.

Deposit and Transaction Phase Correctness We will go through the execution of deposit and transaction phases and show that deposit and transaction phase correctness is satisfied.

When an honest user P_i deposits coins, the deposit will be included in the list of deposits $\mathcal{D}_\Gamma$ on the contract. At the end of the deposit phase the honest operator will wait until k blocks have been published on $\mathcal{L}$. k is the parameter derived from the security parameter κ ensuring that a chain of k blocks cannot be forged by an adversary. The honest operator will provide all blocks since the last phase to the enclave. The enclave then stores all deposits in D and updates the set of parties as $P \cup \{pk_i\}$ for each depositing user.

An honest user P_i submits transactions $((pk_i, pk_j, v, epoch_{P_i}), \sigma_{pk_i})$ only if she has enough balance and also signs the transactions correctly. Furthermore the honest operator stores all valid transactions in the list $\mathcal{T}_\mathcal{O}$ and will forward them to the TEE. Therefore the set of transactions made by the user P_i in this phase is a subset of the transaction which the operator outputs, in other words it holds that $\mathbf{I}[i] = \mathbf{O}[i] = \mathsf{Tx}^s_{P_i}$. Furthermore since the operator is honest, the set of transactions outputted by her are all processed by the TEE. Hence we have $\mathbf{acc}^{new}[i] = \mathbf{acc}^{old}[i] + x + \sum_{(\cdot,\cdot,v)\in \mathsf{Tx}^r_{P_i}} v - \sum_{(\cdot,\cdot,v')\in \mathsf{Tx}^s_{P_i}} v'$ where x is the amount of coins deposited by this user during this epoch. Therefore transaction phase correctness holds

Exit Phase Correctness If an hones user submits an exit message to Γ, the exit is added to the list e_Γ by Γ. At the end of the exit phase an honest operator will forward e_Γ and the chain $\mathbf{b}$ confirming the exit list to the enclave. Note that in case of an honest operator non of the challenges will remain un answered at the end of the exit phase and therefore the operator ill not be announced malicious by Γ. The balance on the enclave is then updated such that $v[i] = 0$ for each exiting user P_i. Furthermore, the enclave will also update the set of participating parties as $P = P \setminus \{pk_i\}$. Therefore, the requirements for exit phase correctness are fulfilled and exit phase correctness holds.

Security

In order to prove that our protocol satisfies security, we show that Deposit Security and Balance Security are satisfied.

Deposit Security According to the enclave program, the operator must provide $t + k$ blocks to the deposit function of the enclave in order to proceed to the next phase. If the operator does so the deposit is processed correctly. However, the operator can refuse to forward the blocks to the enclave which would effectively halt the system since the enclave program would not proceed to the transaction phase or exit phase. This means that the users will not receive their balance at the end of the transaction phase i.e., the message $v_E[i] = ((v_i, epoch_\mathsf{E}, pk_i), \sigma_i)$, since it will not be produced by the enclave and the operator cannot forge such a message (except with negligible probability) because of the unforgeability of the underlying signature scheme. Therefore, the honest users will challenge the operator on-chain and

the operator cannot answer to the challenges of the honest users since she cannot forge a message of the form $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ (except with negligible probability). Finally, the contract will deem the operator malicious and therefore the users will be able to exit their deposit and balance. In other words an honest user P_i will receive $v_i + v_i'$, where v_i is the initial balance of the user at the beginning of this epoch and v_i' is the value P_i deposited in this epoch. Hence, it hold that $(\mathsf{deposit\text{-}returned}, P_i, v) \in \mathbf{O}e[i]$ and $(\mathsf{deposit\text{-}returned}, P_i, v) \in \mathbf{O}e[\Gamma]$ if user P_i made a deposit in this epoch.

Balance Security For balance security we assume that the user P_i is honest and we require that this user receives her whole balance from the contract when exiting. We separate our analysis in to two cases, (1) Honest operator and (2) dishonest operator. Furthermore we do not assume that other users are honest.

Remark 1. The total amount of balances stored on the enclave after a successful transaction phase is never greater than the funds deposited on the Plasma contract Γ.

In order to increase the total amount of balances, the enclave has to process a deposit. To this end at the end of the deposit phase, the operator submits the chain $\bar{\mathbf{b}}$ which consisting of the last $t+k$ blocks on the ledger where the first t blocks of $\bar{\mathbf{b}}$ have to contain the requested deposit list. The security parameter k ensures that it is infeasible for the operator to provide a valid forged chain except with negligible probability. The enclave can hence extract all deposits from $\bar{\mathbf{b}}$ and include them in the deposit list $\mathcal{D}_\Gamma$. The enclave does not include the same list of deposits more than once in the same epoch. Therefore, the total balance on the enclave cannot be greater than the funds on the contract at the end of the transaction phase.

Remark 2. If an honest user requests an exit from the Plasma contract Γ i.e., by submitting the message $(\mathsf{exit}, (v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i))$, she will receive the value she deposited in this epoch as part of the exit procedure.

The contract stores the deposits of the current epoch epoch in the list D. If the operator is honest the deposits of each honest user were included in the Plasma system and are part of the balance value v_i that the user receives when exiting. If the operator is malicious the contract sends additionally the stored deposits of the exiting user via the ledger. Altogether an honest user P_i will receive the correct deposit amount when exiting. In order to show that balance security is satisfied, we analyze the case of an honest and a malicious operator.

Case 1 Honest Operator In case the operator is honest, the user P_i receives the tuple $((v_i, epoch_{\mathsf{E}}, pk_i), \sigma_i)$ at the end of the transaction phase. In order to exit the user forwards this value to the contract which adds the exit request to the list e_Γ. At the end of the exit phase the contract will send the amount v_i to P_i for each exiting user. The correctness of v_i follows from Remark 2 and the fact that TEE processes the transactions made in this epoch correctly. As discussed in remark 1, Γ has enough funds to send v_i to P_i (note that the sum of all balances signed by the TEE is not

greater than the total balance of the contract). Lastly, since the operator is honest she will answer all challenges submitted by other (malicious) users. Altogether we can conclude that v_i coins will be retired to the user on the ledger and $v_i = \mathbf{acc}^{old}[i]$ and therefore balance security is satisfied in the honest operator case.

Case 2 Malicious Operator In case of a malicious operator, the operator may deviate from the CommiTEE protocol.

If the operator does not send the balance value to an honest user P_i, this user will start an exit challenge. Consequently, if the operator responds to the challenge correctly, P_i will exit as in case 1. Therefore balance security is satisfied.

However, if the operator does not respond with valid values to this or any other challenge she is deemed malicious by the contract Γ. In this case the users exit based on their balance from the previous epoch i.e., by submitting $((v_i', epoch_{\mathsf{E}} - 1, pk_i), \sigma_i)$. We note that the users do have this value since otherwise they would have challenged the operator in the previous epoch. Therefore, the amount of coins returned to the users is $v_i' = \mathbf{acc}^{old'}[i] = \mathrm{out}[i]$ and hence balance security is satisfied.

Operator Security Since the operator does not deposit money on the Plasma contract, operator security is trivially satisfied.

Efficiency Since all protocol phases, (namely deposit, transaction and exit phases) of CommiTEE have a fixed constant length on Γ and the honest parties will challenge the operator and exit the system if the operator does not proceed to the next phases on the enclave in any epoch (i.e., by not submitting the transaction list in the transaction phase or the new blocks in the deposit and exit phase to the enclave), the duration of an epoch, δ is constant ($\delta \in O(1)$) and efficiency is satisfied. Note that in the situation described above users will not receive their balance value (since it must be produced by the enclave) and as discussed before honest users will challenge and exit the system. In other words the off-chain execution of the system is synchronized with the epoch length that is enforced by Γ (otherwise the users challenge and exit) and hence the duration of an epoch is constant time.