Ref. Ares(2019)7934126 - 31/12/2019

DS-06-2017: Cybersecurity PPP: Cryptography

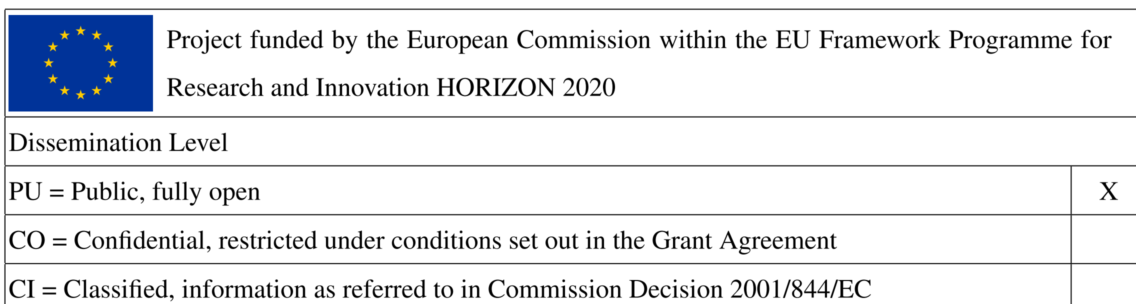
PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D3.2 – Design of Extended Core Protocols

Due date of deliverable: 31 December 2019
Actual submission date: 18 December 2019

Grant agreement number: 780477
Start date of project: 1 January 2018
Revision 1.0

Lead contractor: Guardtime AS
Duration: 36 months



Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020

Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D3.2

Design of Extended Core Protocols

Editor
Toon Segers (TUE)

Contributors
Karim Baghery (UT)
Vincenzo Botta (UNISA)
Michele Ciampi (UEDIN)
Aggelos Kiayias (UEDIN)
Berry Schoenmakers (TUE)
Toon Segers (TUE)
Björn Tackmann (IBM)
Ahto Truu (GT)
Ivan Visconti (UNISA)

Reviewers
Vincenzo Iovino (UNISA)
Toomas Krips (UT)
Panos Louridas (GRNET)
Toon Segers (TUE)
Luisa Siniscalchi (UNISA)

18 December 2019
Revision 1.0

The work described in this document has been conducted within the project PRIViLEDGE, started in January 2018. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477.

The opinions expressed and arguments employed herein do not necessarily reflect the official views of the European Commission.

©Copyright by the PRIViLEDGE Consortium

Contents

1 Executive Summary **1**
1.1 Scope . . . 1
1.1.1 Ledger protocols . . . 1
1.1.2 Secure Multiparty Computation . . . 2
1.1.3 Post-quantum . . . 2
1.2 Impact . . . 2

2 UC-Secure CRS Generation for zk-SNARKs **3**
2.1 Introduction . . . 3
2.2 Preliminaries . . . 5
2.3 Multi-Party CRS Generation . . . 8
2.4 UC-Secure CRS Generation . . . 12
2.5 Secure MPC for NIZKs . . . 14

3 On the Efficiency of Privacy-Preserving Smart Contract Systems **18**
3.1 Introduction . . . 18
3.2 Preliminaries . . . 20
3.2.1 Notations . . . 20
3.2.2 Definitions . . . 21
3.2.3 COCO: a Framework for Constructing UC-secure zk-SNARKs . . . 23
3.3 Efficient UC-secure zk-SNARKs . . . 23
3.3.1 Construction . . . 24
3.3.2 Efficiency . . . 25
3.3.3 Security Proof . . . 25
3.4 On the Efficiency of Privacy Preserving Smart Contract Systems . . . 25

4 Bulletin Board for E-voting **28**
4.1 Introduction . . . 28
4.2 Preliminaries . . . 28
4.3 Our contribution . . . 28
4.3.1 Attacking the liveness of the CS BB system . . . 29
4.3.2 A New Publishing Protocol for the CS BB System . . . 30

5 Asymmetric Distributed Trust **33**
5.1 Introduction . . . 33
5.2 Preliminaries . . . 34
5.3 Recent Related Work . . . 35
5.4 Asymmetric Byzantine Quorum Systems . . . 36
5.4.1 Symmetric Trust . . . 36
5.4.2 Asymmetric Trust . . . 38

5.5 Shared Memory 40
5.5.1 Definitions 41
5.5.2 Protocol with Authenticated Data 41
5.5.3 Double-Write Protocol without Data Authentication 43
5.6 Broadcast 45
5.7 Conclusion 50

6 Proof-of-Stake Sidechains **51**
6.1 Introduction 51
6.2 Preliminaries 54
6.2.1 Our Model 54
6.2.2 Blockchains and Ledgers 54
6.2.3 Underlying Proof-of-Stake Protocols 55
6.3 Defining Security of Pegged Ledgers 56
6.4 Implementing Pegged Ledgers 59
6.4.1 Ad-Hoc Threshold Multisignatures 60
6.4.2 A Concrete Asset $\mathfrak{A}$ 63
6.4.3 The Sidechain Construction 65
6.5 Constructing Ad-Hoc Threshold Multisignatures 75
6.5.1 Plain ATMS 75
6.5.2 Multisignature-based ATMS 75
6.5.3 ATMS From Proofs of Knowledge 76
6.6 Security 76
6.6.1 Assumptions 76
6.6.2 Proof Overview 77
6.6.3 Liveness and Persistence 77
6.6.4 The Firewall Property and MC-Receiving Transactions 78
6.6.5 Firewall Property During Sidechain Failure 79
6.6.6 General Firewall Property 81
6.7 The Diffuse Functionality 84
6.8 Adaptation to Other Proof-of-Stake Blockchains 85
6.8.1 Ouroboros Praos and Ouroboros Genesis 85
6.8.2 Snow White 86
6.8.3 Algorand 86

7 Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake **88**
7.1 Introduction 88
7.2 Protocol Intuition 89
7.2.1 The Foundations of Genesis and Zerocash 90
7.2.2 The Core Protocol 90
7.2.3 Freezing Stake in Zero Knowledge 90
7.2.4 Adaptive Corruptions 91
7.3 The Model 91
7.4 Tools 93
7.4.1 Non-Interactive Zero Knowledge 93
7.4.2 Key-private Forward-Secure Encryption 94
7.4.3 PRFs with unpredictability under malicious keys 96
7.4.4 Equivocal Commitments 96
7.5 The Private Ledger 97
7.5.1 Blinding for Forward-Secure Transactions 99

7.5.2 Leakage for Leader-Based Protocols 100
7.6 The Ouroboros-Crypsinous Protocol 101
7.6.1 Ideal-World Transactions 101
7.6.2 Protocol overview 102
7.6.3 Real-world Transactions 103
7.6.4 Interacting with the Ledger 104
7.6.5 Transaction Validity 106
7.7 Security Analysis 107
7.8 Performance Estimation 111
7.9 Hybrid World Functionalities 112
7.10 The Simulator 114
7.10.1 The Stage 1 Simulator 114
7.10.2 The Stage 2 Simulator 116
7.11 UC Specification of Ouroboros Crypsinous 118
7.11.1 Party Initialization 119
7.11.2 The Staking Procedure 119
7.11.3 The Ledger Maintenance Procedure 120
7.11.4 Submitting Transfer Transactions 121
7.11.5 Submitting Generic Transactions 122
7.11.6 Reading the Ledger State 122
7.12 NP Statements 123
7.13 Protocol Assumptions Encoded as a Wrapper 124
7.14 Construction NIZKs via SNARKs 125
7.14.1 Proof of UC-Emulation 126
7.15 Key-Private Forward-Secure Encryption 130

8 Privacy Threats Exploiting Smart Contracts and Forks **134**
8.1 Attack to the Zero-Knowledge Property of GG-NIZK [GG17] 134
8.2 Attacking and Repairing Smart Contracts on Forking Blockchains 136

9 Verifiable MPC with Blockchain **143**
9.1 Introduction 143
9.2 Preliminaries 144
9.2.1 The Notion of Outsourcing 144
9.2.2 Secure Multiparty Computation 145
9.2.3 Verifiable Computation with Zero-Knowledge 145
9.2.4 Blockchain 147
9.3 Recent Related Work in Verifiable Multiparty Computation 147
9.3.1 Trinocchio: Verifiable Computation on Private Inputs 147
9.3.2 Geppetri: Reusable Setup for Different Computations with Adaptive zk-SNARKs 149
9.4 Verifiable MPC with Blockchain 149
9.4.1 Optional: Use Smart Contract to Verify Correctness Proof 150
9.4.2 The Billionaires' Problem: Solution Sketch 151
9.4.3 Main Protocols to Compute the Top 400: Verifiable Secure Filtering and Sorting 151
9.4.4 Verifiable Secure Comparison 152
9.4.5 The Billionaires' Problem: Next steps and Implementation Framework 153
9.5 Conclusion 153

10 Hash Based Server Assisted Signatures **154**
10.1 Background 154
10.2 Blockchain Backed Key State Management 155
10.3 Forward-Resistant Tag Systems 155

11 Conclusion **157**

Chapter 1

Executive Summary

This document presents the design of core protocols for distributed ledgers with a focus on privacy and security. Contributions were developed by PRIViLEDGE partners during the first 24 months of the project. The contributions presented in the following chapters include both peer-reviewed and ongoing research work.

This section briefly summarizes the contents of the subsequent chapters. Each chapter discusses a contribution's relevance, preliminaries, new results and technical details.

1.1 Scope

The document covers three main protocol categories:

- Ledger protocols;
- Secure multiparty computation (MPC) protocols;
- Protocols in the post-quantum setting.

1.1.1 Ledger protocols

Chapter 2 formalizes **universal composability (UC) security of the trusted setup of SNARKs with MPC**. Constructing efficient zk-SNARKs with minimized trust assumptions is an important research topic, as many DLT protocols and applications rely on zk-SNARKs. This result shows that if we use the UC-secure MPC protocol explained in Chapter 2 for CRS generation of zk-SNARK proposed in [ABLZ17], the prover and verifier need to trust only a single party among parties of the MPC protocol. The proposed protocol can be used to mitigate the trust in applications that use UC-secure SNARKs; e.g. Hawk [KMS+15] and Gyges [JKS16].

Chapter 3 improves the **efficiency of privacy-preserving smart contract systems** such as Hawk [KMS+15] and Gyges [JKS16], by improving the efficiency of the underlying zk-SNARK. Improving the efficiency of zk-SNARKs in these systems is relevant because the performance of these systems greatly depends on it. The proposed construction can be of independent interest for all applications that require a UC-secure NIZK or zk-SNARK.

Chapter 4 presents an **attack against the Bulletin Board construction by Culnane and Schneider** [CS14b] where an attacker can prevent termination when less than $N/3$ of the N bulletin board peers are corrupted (contrary to what is claimed in [CS14b]). Chapter 4 then proposes a scheme to mitigate this attack.

Chapter 5 presents a formal introduction of **asymmetric Byzantine quorum systems**, extending standard Byzantine quorum (or consensus) systems. Furthermore, Chapter 5 presents protocols that adopt this asymmetric quorum system, specifically for shared read-write registers and broadcasting. To illustrate its relevance: The broadcasting protocol is, for example, relevant for "federated voting" in the public Stellar blockchain.[1]

[1] See https://www.stellar.org or https://medium.com/interstellar/understanding-the-stellar-consensus-protocol-423409aad32e for an informal explanation of Stellar's federated voting protocol

Chapter 6 presents a novel formalization of **Proof-of-Stake (PoS) Sidechains**. Then, Chapter 7 presents a formal model for a **privacy-preserving PoS-based distributed ledger** in the universal composability (UC) setting, including a protocol that realizes this new type of ledger. Both chapters present very relevant extensions of classical PoS blockchains, addressing key pain-points of interoperability, scalability, upgradability and privacy.

Chapter 8 presents two attacks related to applications of distributed ledger technology. The first attack concerns the adversarial use of smart contracts to **break the zero-knowledge property of the non-interactive zero-knowledge (NIZK) argument** proposed by [GG17] without corrupting any party. The second attack concerns the adversarial use of forks in blockchains in order to **break the security of a smart contract that is supposed to securely implement a two or multi-party functionality.**

1.1.2 Secure Multiparty Computation

Chapter 9 introduces the notion of **verifiable MPC with blockchain**. Verifiable MPC allows a client (or multiple clients) to outsource a computation in a privacy-preserving way, while enabling public verifiability of correctness. The protocol in Chapter 9 enables a client to use verifiable input data (via cryptographic commitments), outsource expensive computation to an off-chain network of workers (in a privacy-preserving way), verify the result on-chain and reuse the result for other use-cases.

1.1.3 Post-quantum

Chapter 10 presents a contribution on **server assisted digital signature schemes** whose security is **based on hash functions**. The server assisted nature of these schemes is a good match for distributed ledger systems. According to current knowledge, hash functions are resilient to attacks by quantum computers. While formal analysis of the new schemes in quantum setting is still ongoing, we expect them to remain secure even against quantum adversaries, in contrast with the currently prevalent schemes based on integer factoring and discrete logarithm problems.

1.2 Impact

The impact of these PRIViLEDGE contributions is illustrated by the fact that most have been peer-reviewed and/or accepted at conferences very recently.[2] The table below presents the outlets that have accepted these contributions.

Contribution	Outlet
UC-Secure CRS Generation for zk-SNARKs	Africacrypt 2019
On the Efficiency of Privacy-Preserving Smart Contract Systems	Africacrypt 2019
Bulletin Board for E-voting	SCN 2018
Asymmetric Distributed Trust	DISC 2019
Proof-of-Stake Sidechains	IEEE Symposium on Security and Privacy 2019
Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake	IEEE Symposium on Security and Privacy 2019
Hash Based Server Assisted Signatures	ACM SIGPLAN CPP 2020

[2] For *Asymetric Distributed Trust*, a brief announcement was presented at DISC 2019.

Chapter 2

UC-Secure CRS Generation for zk-SNARKs

2.1 Introduction

A zero-knowledge argument is a cryptographic protocol between a prover and a verifier where the objective is to prove the validity of some statement while not leaking any other information. In particular, such an argument should be *sound* (it should be impossible to prove false statements) and *zero-knowledge* (the only leaked information should be the validity of the statement). Practical applications often require a non-interactive zero-knowledge (NIZK) argument where the prover outputs a single message which can be checked by many different verifiers.

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) are particularly efficient instantiations of NIZK, and have thus found numerous application ranging from verifiable computation [PHGR13] to privacy-preserving cryptocurrencies [KMS+15] and privacy-preserving smart contracts [KMS+15]. In most of such zk-SNARKs (see, e.g., [Gro10, Lip12, GGPR13, PHGR13, DFGK14, Gro16]), the verifier's computation is dominated by a small number of exponentiations and pairings in a bilinear group, while the argument consists of a small number of group elements. Importantly, a zk-SNARK exists for any NP-language.

One drawback in the mentioned pairing-based zk-SNARKs is their reliance on the strong common reference string (CRS) model. It assumes that in the setup phase of the protocol a trusted party publishes a CRS, sampled from some specialized distribution, while not leaking any side information. Subverting the setup phase can make it easy to break the security, e.g., leaking a CRS trapdoor makes it trivial to prove false statements. This raises the obvious question of how to apply zk-SNARKs in practice without completely relying on a single trusted party. The issue is further amplified since in all of the mentioned zk-SNARKs, one has to generate a new CRS each time the relation changes.

Reducing trust on CRS generation is indeed a long-standing open question. Several different approaches for this are known, but each one has its own problems. Some recent papers [BCG+15, BGG17, BGM17] have proposed efficient CRS-generation using multi-party computation protocols, where only 1 out of N_p parties (where N_p denotes the number of parties participated in the MPC protocol) has to be honest, for a large class of known zk-SNARKs (in fact, most of the efficient pairing-based zk-SNARKs belong to this class, possibly after the inclusion of a small number of new elements to their CRSs) for which the CRS can be computed by a fixed well-defined class $\mathcal{C}^{\mathsf{S}}$ of circuits. Following [BCG+15], we will call this class of zk-SNARKs $\mathcal{C}^{\mathsf{S}}$-*SNARKs*. However, the CRS-generation protocols of [BCG+15, BGG17, BGM17] have the following two weaknesses:

They are not secure in the universal composability (UC) setting [Can01]. Hence, they might not be secure while running in parallel with other protocols, as is often the case in real life scenarios. Moreover, some systems require a UC-secure NIZK [KMS+15, JKS16], but up to now their CRS is still be generated in a standalone setting. We note that [BCG+15, BGM17] do prove some form of simulatability but not for full UC-security. Protocol of [BGG17] is for one specific zk-SNARK.

All use the random oracle model and [BGG17, BGM17] additionally use knowledge assumptions. Non-

falsifiable assumptions [Nao03] (e.g., knowledge assumptions) and the random oracle model are controversial (in particular, the random oracle model is uninstantiable [CGH98, GK03] and thus can only be thought of as a heuristic), and it is desirable to avoid them in situations where they are not known to circumvent impossibility results. Importantly, construction of zk-SNARKs under falsifiable assumptions is impossible [GW11] and hence they do rely on non-falsifiable assumptions but usually not on the random oracle model. Relying on the random oracle model in the setup phase means that the complete composed system (CRS-generation protocol + zk-SNARK) relies on both random oracle model and non-falsifiable assumptions. Hence, we end up depending on two undesirable assumptions rather than one.

Updatable CRS [GKM+18] is another recent solution to the problem. Essentially, this can be viewed as a single round MPC protocol where each party needs to participate just once in the CRS computation. Current zk-SNARKs in updatable CRS model [GKM+18, MBKM19] are still less efficient, than the state-of-the-art non-updatable counterparts like the zk-SNARK by Groth [Gro16].

As a different approach, in order to minimize the trust of NIZKs in the setup phase, Bellare *et al.* [BFS16] defined the notion of subversion-resistance, which guarantees that a security property (like soundness) holds even if the CRS generators are all malicious. As proven in [BFS16], achieving subversion-soundness and (even non-subversion) zero knowledge at the same time is impossible for NIZK arguments. On the other hand, one can construct subversion-zero knowledge (Sub-ZK) and sound NIZK arguments. Abdolmaleki *et al.* [ABLZ17] showed how to design efficient Sub-ZK SNARKs: essentially, a zk-SNARK can be made Sub-ZK by constructing an efficient public CRS-verification algorithm CV that guarantees the well-formedness of its CRS. In particular, [ABLZ17] did this for the most efficient known zk-SNARK by Groth [Gro16] after inserting a small number of new elements to its CRS. Fuchsbauer [Fuc18] proved that Groth's zk-SNARK (with a slightly different simulation) is Sub-ZK even without changing its CRS. Recently, Baghery [Bag19b] showed that one can achieve Sub-ZK and simulation knowledge soundness in a recently proposed variation of Groth's zk-SNARK [AB19] at the same time.

Contribution. In [ABL+19b], we propose a new UC-secure multi-party CRS-generation protocol for $\mathcal{C}^{\mathsf{S}}$-SNARKs that crucially relies only on *falsifiable assumptions* and *does not require a random oracle*. Conceptually, the new protocol follows similar ideas as the protocol of [BCG+15], but it does not use any proofs of knowledge. Instead, we use a discrete logarithm extractable (DL-extractable) UC commitment functionality $\mathcal{F}_{\mathsf{dlmcom}}$ that was recently defined by Abdolmaleki *et al.* [ABL+19a][1]. A DL-extractable commitment scheme allows to commit to a field element x and open to the group element g^x. Since $\mathcal{F}_{\mathsf{dlmcom}}$ takes x as an input, the committer must know x and thus x can be extracted by the UC-simulator. As we will show, this is sufficient to prove UC-security of the new CRS-generation protocol.

In addition, we show that the Sub-ZK SNARK of [ABLZ17] is a Sub-ZK $\mathcal{C}^{\mathsf{S}}$-SNARK after just adding some more elements to its CRS. We also improve the efficiency of the rest of the CRS-generation protocol by allowing different circuits for each group, considering special multiplication-division gates, and removing a number of NIZK proofs that are used in [BCG+15]. Like in the previous CRS-generation protocols [BCG+15, BGG17, BGM17], soundness and zero-knowledge will be guaranteed as long as 1 out of N_p parties participating in the CRS generation is honest. If SNARK is also Sub-ZK [ABLZ17, Fuc18], then zero-knowledge is guaranteed even if all N_p parties are dishonest, given that the prover executes a public CRS verification algorithm.

Since it is impossible to construct UC commitments in the standard model [CF01], the new UC-secure CRS-generation protocol necessarily relies on some trust assumption. The DL-extractable commitment scheme of [ABL+19a] is secure in the registered public key (RPK) model[2] that is a weaker trust model than the CRS model, as in the CRS model all parties require to trust a single party. However, we stay agnostic to the concrete implementation of $\mathcal{F}_{\mathsf{dlmcom}}$, proving the security of the CRS-generation protocol in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model.

[1] Note that our proposed approach can be used to generate the CRS of either non-UC-secure or UC-secure zk-SNARKs.

[2] In the RPK model, each party registers his public key with an authority of his choosing. It is assumed that even authorities of untrusted parties are honest to the extent that they verify the knowledge (e.g., by using a standalone ZK proof) of the corresponding secret key.

Thus, the trust assumption of the CRS-generation protocol is directly inherited from the trust assumption of the used DL-extractable commitment scheme. Constructing DL-extractable commitment schemes in a weaker model like the random string model or the multi-string model is an interesting open question.

It is possible to realize any UC-functionality, including $\mathcal{F}_{\mathsf{dlmcom}}$, in even weaker common random string model [CLOS02] or in the multi-string model [GO07]. First one could be easily set up in practice by extracting a random string from some natural source (see [CPS07] for further details) and the second model assumes that majority of parties generate a uniformly random string. Although current implementations of $\mathcal{F}_{\mathsf{dlmcom}}$ in the previous two models would be relatively inefficient compared to the RPK model implementation, it would still be suitable for our purpose as we make only a small constant number of commitments (e.g., Groth's zk-SNARK [Gro16] uses five different trapdoors, and hence $\mathcal{F}_{\mathsf{dlmcom}}$ has to be called five times; moreover, [BGM17] claims only four trapdoors are needed). Note that CRS-s of known *efficient* $\mathcal{C}^{\mathsf{S}}$-SNARKs, with a few exceptions, contain $\Omega(\mathsf{n})$ group elements, where n is the circuit size (e.g., in the last CRS generation of Zcash [BCG+14], $\mathsf{n} \approx 2\,000\,000$ [3]). Hence, even a relatively inefficient DL-extractable commitment scheme (that only has to be called once per CRS trapdoor) will not be the bottleneck in the CRS-generation protocol.

We proceed as follows. First, we describe an ideal functionality $\mathcal{F}_{\mathsf{mcrs}}$, an explicit multi-party version of the CRS generation functionality. Intuitively (the real functionality is slightly more complicated), first, N_p key-generators $\mathscr{G}_i$ send to $\mathcal{F}_{\mathsf{mcrs}}$ their shares of the trapdoors, s.t. the shares of the honest parties are guaranteed to be uniformly random. Second, $\mathcal{F}_{\mathsf{mcrs}}$ combines the shares to create the trapdoors and the CRS, and then sends the CRS to each $\mathscr{G}_i$. We propose a protocol $\mathsf{K}_{\mathsf{mcrs}}$ that UC-realizes $\mathcal{F}_{\mathsf{mcrs}}$ in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model, i.e., assuming the availability of a UC-secure realization of $\mathcal{F}_{\mathsf{dlmcom}}$. In $\mathsf{K}_{\mathsf{mcrs}}$, the parties $\mathscr{G}_i$ first $\mathcal{F}_{\mathsf{dlmcom}}$-commit to their individual share of each trapdoor. After opening the commitments, $\mathscr{G}_i$ compute crs by combining their shares with a variation of the protocol from [BCG+15]. The structure of this part of the protocol makes it possible to publicly check that it was correctly followed.

Next, we prove that a $\mathcal{C}^{\mathsf{S}}$-SNARK that is complete, sound, and Sub-ZK in the CRS model is also complete, sound, and Sub-ZK in the $\mathcal{F}_{\mathsf{mcrs}}$-hybrid model. Sub-ZK holds even if all CRS creators were malicious, but for soundness we need at least one honest party. We then show that the Sub-ZK secure version [ABLZ17, Fuc18] of the most efficient known zk-SNARK by Groth [Gro16] remains sound and Sub-ZK if the CRS has been generated by using $\mathsf{K}_{\mathsf{mcrs}}$. The main technical issue here is that since Groth's zk-SNARK is not $\mathcal{C}^{\mathsf{S}}$-SNARK (see 2.3), we need to add some new elements to its CRS and then reprove its soundness against an adversary who is given access to the new CRS elements. We note that Bowe et al. [BGM17] proposed a different modification of Groth's zk-SNARK together with a CRS-generation protocol, but under strong assumptions of random beacon model, random oracle model, and knowledge assumptions. Role of the commitment in their case is substituted with a random beacon which in particular means that they do not need to fix parties in the beginning of the protocol.

We constructed a UC-secure CRS-generation protocol $\mathsf{K}_{\mathsf{mcrs}}$ in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model for any $\mathcal{C}^{\mathsf{S}}$-SNARK and in particular proved that a small modification of Groth's zk-SNARK remains secure when its CRS is generated with $\mathsf{K}_{\mathsf{mcrs}}$. Moreover, the resulting CRS-generation protocol is essentially as efficient as the prior protocols from [BCG+15,BGG17,BGM17]. However, (i) we proved the UC-security of the new CRS-generation protocol, and (ii) the new protocol is falsifiable, i.e., it does not require either the random oracle model or any knowledge assumption.

2.2 Preliminaries

Let PPT denote probabilistic polynomial-time. Let $\kappa \in \mathbb{N}$ be the information-theoretic security parameter, in practice, e.g., $\kappa = 128$. All adversaries will be stateful. For an algorithm $\mathcal{A}$, let $\mathrm{im}(\mathcal{A})$ be the image of $\mathcal{A}$, i.e., the set of of valid outputs of $\mathcal{A}$, let $\mathsf{RND}\mathcal{A}$ denote the random tape of $\mathcal{A}$, and let $r \leftarrow \mathsf{RND}\mathcal{A}$ denote sampling of a randomizer r of sufficient length for $\mathcal{A}$'s needs. By $y \leftarrow \mathcal{A}(x; r)$ we denote that $\mathcal{A}$, given an input x and a randomizer r, outputs y. We denote by negl an arbitrary negligible function, and by $\mathsf{poly}(\kappa)$ an arbitrary

[3] See https://www.zfnd.org/blog/conclusion-of-powers-of-tau/

polynomial function. $A \approx_c B$ means that distributions A and B are computationally indistinguishable. We write $x \leftarrow \mathscr{D}$ if x is sampled according to distribution $\mathscr{D}$ or uniformly in case $\mathscr{D}$ is a set. By $\mathtt{Supp}(\mathscr{D})$ we denote the set of all elements in $\mathscr{D}$ that have non-zero probability.

Assume that $\mathscr{G}_i$ are different parties of a protocol. Following previous work [BCG+15], we will make the following assumptions about the network and the adversary. It is possible that the new protocols can be implemented in the asynchronous model but this is out of scope of the current paper.

Synchronicity assumptions: We assume that the computation can be divided into clearly divided rounds. As it is well-known, synchronous computation can be simulated, assuming bounded delays and bounded time-drift. For the sake of simplicity, we omit formal treatment of UC-secure synchronous execution, see [KMTZ13] for relevant background.

Authentication: we assume the existence of an authenticated broadcast between the parties. In particular, (i) if an honest party broadcasts a message, we assume that all parties (including, in the UC-setting, the simulator) receive it within some delay, and (ii) an honest party $\mathscr{G}_j$ accepts a message as coming from $\mathscr{G}_i$ only if it was sent by $\mathscr{G}_i$.

Covertness: We assume that an adversary in the multi-party protocols is covert, i.e., it will not produce outputs that will not pass public verification algorithms. In the protocols we write that honest parties will abort under such circumstances, but in the proofs we assume that adversary will not cause abortions.

For pairing-based groups we will use additive notation together with the bracket notation, i.e., in group $\mathbb{G}_1$, $[a]_1 = a\,[1]_1$, where $[1]_1$ is a fixed generator of $\mathbb{G}_1$. A deterministic *bilinear group generator* $\mathsf{Pgen}(1^\kappa)$ returns $\mathsf{p} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2)$, where p (a large prime) is the order of cyclic abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$, and $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficient non-degenerate bilinear pairing, s.t. $\hat{e}([a]_1, [b]_2) = [ab]_T$. Denote $[a]_1 \bullet [b]_2 = \hat{e}([a]_1, [b]_2)$; this extends to vectors in a natural way. Occasionally we write $[a]_z \bullet [b]_{3-z}$ for $z \in \{1, 2\}$ and ignore the fact that for $z = 2$ it should be written $[b]_{3-z} \bullet [a]_z$. Let $[a]_\star := ([a]_1, [a]_2)$. As in [BFS16], we will implicitly assume that p is generated deterministically from κ; in particular, the choice of p cannot be subverted.

UC Security. We work in the standard universal composability framework of Canetti [Can01] with static corruptions of parties. The UC framework defines a PPT environment machine $\mathcal{Z}$ that oversees the execution of a protocol in one of two worlds. The "ideal world" execution involves "dummy parties" (some of whom may be corrupted by an ideal adversary/simulator Sim) interacting with a functionality $\mathcal{F}$. The "real world" execution involves PPT parties (some of whom may be corrupted by a PPT real world adversary $\mathcal{A}$) interacting only with each other in some protocol π. We refer to [Can01] for a detailed description of the executions, and a definition of the real world ensemble $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ and the ideal world ensemble $\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim}^\mathcal{A},\mathcal{Z}}$. A protocol π *UC-securely computes* $\mathcal{F}$ if there exists a PPT Sim such that for every non-uniform PPT $\mathcal{Z}$ and PPT $\mathcal{A}$, $\{\mathsf{IDEAL}_{\mathcal{F},\mathsf{Sim}^\mathcal{A},\mathcal{Z}}(\kappa, \mathsf{x})\}_{\kappa\in\mathbb{N},\mathsf{x}\in\{0,1\}^*} \approx_c \{\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, \mathsf{x})\}_{\kappa\in\mathbb{N},\mathsf{x}\in\{0,1\}^*}$.

The importance of this definition is a composition theorem that states that any protocol that is universally composable is secure when run concurrently with many other arbitrary protocols; see [Can01] for discussions and definitions.

CRS functionality. The CRS model UC functionality $\mathcal{F}_{\mathsf{crs}}^{\mathscr{D},f}$ parameterized by a distribution $\mathscr{D}$ and a function f intuitively works as follows. Functionality samples a trapdoor $\mathtt{tc}$ from $\mathscr{D}$, computes $\mathsf{crs} = f(\mathtt{tc})$, and stores crs after a confirmation from the simulator. Subsequently on each retrieval query $(\mathtt{retrieve}, \mathsf{sid})$ it responds by sending $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$. For full details see Fig. 2.1.

$\mathcal{F}_{\mathsf{crs}}^{\mathscr{D},f}$ is parametrized by a distribution $\mathscr{D}$ and a function f. It proceeds as follows, running with parties $\mathscr{G}_i$ and an adversary Sim.
CRS generation: Sample $\mathsf{tc} \leftarrow \mathscr{D}$; Set $\mathsf{crs} \leftarrow f(\mathsf{tc})$; Send $(\mathtt{crsOK?}, \mathsf{sid}, \mathsf{crs})$ to Sim; If Sim returns $(\mathtt{crsOK}, \mathsf{sid})$ then store $(\mathsf{sid}, \mathsf{crs})$.
Retrieval: upon receiving $(\mathtt{retrieve}, \mathsf{sid})$ from $\mathscr{G}_i$: If $(\mathsf{sid}, \mathsf{crs})$ is recorded for some crs then send $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$ to $\mathscr{G}_i$. Otherwise, ignore the message.

Figure 2.1: Functionality $\mathcal{F}_{\mathsf{crs}}^{\mathscr{D},f}$

$\mathcal{F}_{\mathsf{dlmcom}}$, parametrized by $\mathcal{M} = \mathbb{Z}_p$ and group $\mathbb{G}_\iota$, interacts with $\mathscr{G}_1, \ldots, \mathscr{G}_{\mathsf{N}_p}$ as follows.

Upon receiving $(\mathtt{commit}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, \mathscr{G}_j, m)$ from $\mathscr{G}_i$, where $m \in \mathbb{Z}_p$: if a tuple $(\mathsf{sid}, \mathsf{cid}, \cdots)$ with the same $(\mathsf{sid}, \mathsf{cid})$ was previously recorded, do nothing. Otherwise, record $(\mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, \mathscr{G}_j, m)$ and send $(\mathtt{rcpt}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, \mathscr{G}_j)$ to $\mathscr{G}_j$ and Sim.

Upon receiving $(\mathtt{open}, \mathsf{sid}, \mathsf{cid})$ from $\mathscr{G}_i$, proceed as follows: if a tuple $(\mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, \mathscr{G}_j, m)$ was previously recorded then send $(\mathtt{open}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, \mathscr{G}_j, y \leftarrow [m]_\iota)$ to $\mathscr{G}_j$ and Sim. Otherwise do nothing.

Figure 2.2: Functionality $\mathcal{F}_{\mathsf{dlmcom}}$ for $\iota \in \{1, 2\}$

DL-extractable UC Commitment. Abdolmaleki et al. [ABL+19a] recently proposed a discrete logarithm extractable (DL-extractable) UC-commitment scheme. Differently from the usual UC-commitment, a committer will open the commitment to $[m]_1$, but the functionality also guarantees that the committer knows x. Hence, in the UC security proof it is possible to extract the discrete logarithm of $[m]_1$. Formally, the ideal functionality $\mathcal{F}_{\mathsf{dlmcom}}$ takes m as a commitment input (hence the user must know m), but on $\mathtt{open}$ signal only reveals $[m]_1$. See Fig. 2.2. We refer to [ABL+19a] for a known implementation of $\mathcal{F}_{\mathsf{dlmcom}}$ in the RPK model.

Non-interactive zero-knowledge. Let $\mathcal{R}$ be a relation generator, such that $\mathcal{R}(1^\kappa)$ returns a polynomial-time decidable binary relation $\mathbf{R} = \{(\mathsf{x}, \mathsf{w})\}$. Here, x is the statement and w is the witness. We assume that κ is explicitly deducible from the description of $\mathbf{R}$. The relation generator also outputs auxiliary information $\xi_{\mathbf{R}}$ that will be given to the honest parties and the adversary. As in [Gro16, ABLZ17], $\xi_{\mathbf{R}}$ is the value returned by $\mathsf{Pgen}(1^\kappa)$. Because of this, we also give $\xi_{\mathbf{R}}$ as an input to the honest parties; if needed, one can include an additional auxiliary input to the adversary. Let $\mathbf{L}_{\mathbf{R}} = \{\mathsf{x} : \exists \mathsf{w}, (\mathsf{x}, \mathsf{w}) \in \mathbf{R}\}$ be an NP-language.

A (subversion-resistant) *non-interactive zero-knowledge argument system* [ABLZ17] Ψ for $\mathcal{R}$ consists of six PPT algorithms:

CRS trapdoor generator: K_{tc} is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}) \in \mathrm{im}(\mathcal{R}(1^\kappa))$, outputs a *CRS trapdoor* tc. Otherwise, it outputs $\perp$.

CRS generator: $\mathsf{K}_{\mathsf{crs}}$ is a *deterministic* algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{tc})$, where $(\mathbf{R}, \xi_{\mathbf{R}}) \in \mathrm{im}(\mathcal{R}(1^\kappa))$ and $\mathsf{tc} \in \mathrm{im}(\mathsf{K}_{\mathsf{tc}}(\mathbf{R}, \xi_{\mathbf{R}})) \setminus \{\perp\}$, outputs crs. Otherwise, it outputs $\perp$. We distinguish three parts of crs: $\mathsf{crs}_{\mathsf{P}}$ (needed by the prover), $\mathsf{crs}_{\mathsf{V}}$ (needed by the verifier), and $\mathsf{crs}_{\mathsf{CV}}$ (needed by CV algorithm).

CRS verifier: CV is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs})$, returns either 0 (the CRS is ill-formed) or 1 (the CRS is well-formed).

Prover: P is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})$, where $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$, outputs an argument π. Otherwise, it outputs $\perp$.

Verifier: V is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi)$, returns either 0 (reject) or 1 (accept).

Simulator: Sim is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \mathsf{tc}, \mathsf{x})$, outputs an argument π.

We also define the CRS generation algorithm $\mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}})$ that first sets $\mathsf{tc} \leftarrow \mathsf{K}_{\mathsf{tc}}(\mathbf{R}, \xi_{\mathbf{R}})$ and then outputs $\mathsf{crs} \leftarrow \mathsf{K}_{\mathsf{crs}}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{tc})$.

Ψ is *perfectly complete for* $\mathcal{R}$, if for all κ, $(\mathbf{R}, \xi_{\mathbf{R}}) \in RANGE(\mathcal{R}(1^\kappa))$, $\mathsf{tc} \in RANGE(\mathsf{K}_{\mathsf{tc}}(\mathbf{R}, \xi_{\mathbf{R}})) \setminus \{\perp\}$, and $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$,

$$\Pr\left[\mathsf{crs} \leftarrow \mathsf{K}_{\mathsf{crs}}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{tc}) : \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \mathsf{P}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})) = 1\right] = 1 \enspace .$$

Ψ is computationally adaptively *knowledge-sound for* $\mathcal{R}$ [Gro16], if for every non-uniform PPT $\mathcal{A}$, there exists a non-uniform PPT extractor $\mathsf{Ext}_{\mathcal{A}}$, s.t. $\forall\ \kappa$,

$$\Pr\left[\begin{array}{l}(\mathbf{R}, \xi_{\mathbf{R}}) \leftarrow \mathcal{R}(1^\kappa), (\mathsf{crs}, \mathsf{tc}) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}), r \leftarrow_r \mathsf{RND}\mathcal{A}, \\ (\mathsf{x}, \pi) \leftarrow \mathcal{A}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}; r), \mathsf{w} \leftarrow \mathsf{Ext}_{\mathcal{A}}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}; r) : \\ (\mathsf{x}, \mathsf{w}) \not\in \mathbf{R} \wedge \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi) = 1\end{array}\right] \approx_\kappa 0 \enspace .$$

Here, $\xi_{\mathbf{R}}$ can be seen as a common auxiliary input to $\mathcal{A}$ and $\mathsf{Ext}_{\mathcal{A}}$ that is generated by using a benign [**?**] relation generator; we recall that we just think of $\xi_{\mathbf{R}}$ as being the description of a secure bilinear group.

Ψ is *statistically unbounded ZK for* $\mathcal{R}$ [Gro06], if for all κ, all $(\mathbf{R}, \xi_{\mathbf{R}}) \in RANGE(\mathcal{R}(1^\kappa))$, and all computationally unbounded $\mathcal{A}$, $\varepsilon_0^{unb} \approx_\kappa \varepsilon_1^{unb}$, where

$$\varepsilon_b^{unb} = \Pr[(\mathsf{crs}, \mathsf{tc}) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}) : \mathcal{A}^{o_b(\cdot,\cdot)}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}) = 1] \enspace .$$

Here, the oracle $o_0(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \not\in \mathbf{R}$, and otherwise it returns $\mathsf{P}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})$. Similarly, $o_1(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \not\in \mathbf{R}$, and otherwise it returns $\mathsf{Sim}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \mathsf{tc}, \mathsf{x})$. Ψ is *perfectly unbounded ZK for* $\mathcal{R}$ if one requires that $\varepsilon_0^{unb} = \varepsilon_1^{unb}$.

Ψ is *statistically unbounded Sub-ZK for* $\mathcal{R}$, if for any non-uniform PPT subverter X there exists a non-uniform PPT $\mathsf{Ext}_{\mathsf{X}}$, such that for all κ, $(\mathbf{R}, \xi_{\mathbf{R}}) \in RANGE(\mathcal{R}(1^\kappa))$, and computationally unbounded $\mathcal{A}$, $\varepsilon_0^{unb} \approx_\kappa \varepsilon_1^{unb}$, where

$$\varepsilon_b^{unb} = \Pr\left[\begin{array}{l}r \leftarrow_r \mathsf{RNDX}, (\mathsf{crs}, \xi_{\mathsf{X}}) \leftarrow \mathsf{X}(\mathbf{R}, \xi_{\mathbf{R}}; r), \mathsf{tc} \leftarrow \mathsf{Ext}_{\mathsf{X}}(\mathbf{R}, \xi_{\mathbf{R}}; r) : \\ \mathsf{CV}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}) = 1 \wedge \mathcal{A}^{o_b(\cdot,\cdot)}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \mathsf{tc}, \xi_{\mathsf{X}}) = 1\end{array}\right] \enspace .$$

Here, the oracle $o_0(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \not\in \mathbf{R}$, and otherwise it returns $\mathsf{P}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})$. Similarly, $o_1(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \not\in \mathbf{R}$, and otherwise it returns $\mathsf{Sim}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \mathsf{tc}, \mathsf{x})$. Ψ is *perfectly unbounded Sub-ZK for* $\mathcal{R}$ if one requires that $\varepsilon_0^{unb} = \varepsilon_1^{unb}$.

Intuitively the previous definition says that an argument is Sub-ZK when for any untrusted (efficient) CRS generator X, some well-formedness condition $\mathsf{CV}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}) = 1$ implies that X knows a trapdoor which would allow him to simulate the proof. Hence, to protect privacy from malicious CRS generators, the prover just needs to verify that the CRS satisfies the CV algorithm.

Finally, a non-interactive argument system is *succinct* if the argument length is polynomial in κ and the verifier runs in time polynomial in $\kappa + |\mathsf{x}|$.

2.3 Multi-Party CRS Generation

Recently, [BCG+15, BGG17, BGM17] proposed several multi-party CRS-generation protocols for SNARKs. In particular, [BCG+15] proposes a specific class of arithmetic circuits $\mathcal{C}^{\mathsf{S}}$, shows how to evaluate $\mathcal{C}^{\mathsf{S}}$-circuits in an MPC manner, and claims that $\mathcal{C}^{\mathsf{S}}$-circuits can be used to compute CRS-s for a broad class of SNARKs, which here we call them $\mathcal{C}^{\mathsf{S}}$-SNARKs. The CRS of each $\mathcal{C}^{\mathsf{S}}$-SNARK is an output of some $\mathcal{C}^{\mathsf{S}}$-circuit taken into exponent. The input of such circuit is the CRS trapdoor. In the following, we review and modify the framework of [BCG+15] while slightly redefining the class $\mathcal{C}^{\mathsf{S}}$ and the CRS-generation protocol.

$\mathcal{C}^{\mathsf{S}}$-Circuits. For an arithmetic circuit C over a field $\mathbb{F}$, denote by $\mathsf{wires}(\mathsf{C})$ and $\mathsf{gates}(\mathsf{C})$ the set of wires and gates of C (each gate can have more than one output wire), and by $\mathsf{inputs}(\mathsf{C}), \mathsf{outputs}(\mathsf{C}) \subset \mathsf{wires}(\mathsf{C})$ the set of input and output wires of C. There can also be wires with hard-coded constant values, but these are not considered to be part of $\mathsf{inputs}(\mathsf{C})$. The size of C is $|\mathsf{inputs}(\mathsf{C})| + |\mathsf{gates}(\mathsf{C})|$. For a wire w we denote the value on the wire by $\bar{w}$; this notation also extends to tuples, say, $\overline{\mathsf{inputs}}(\mathsf{C})$ denotes the tuple of values of $\mathsf{inputs}(\mathsf{C})$.

For a gate g, $\mathsf{output}(g) = w$ is the output wire and the tuple of all input wires is denoted by $\mathsf{inputs}(g)$. Let g_w be the gate with $w = \mathsf{output}(g_w)$. We consider circuits with *addition* and *multiplication-division* gates. For an addition gate ($\mathsf{type}(g) = \mathsf{add}$), $\mathsf{inputs}(g) = (w_1, \ldots, w_f)$, $\mathsf{coeffs}(g) = (a_0, a_1, \ldots, a_f)$, and it outputs a value $\bar{w} = a_0 + \sum_{j=1}^{f} a_j \bar{w}_j$. For a multdiv gate ($\mathsf{type}(g) = \mathsf{multdiv}$), $\mathsf{inputs}(g) = (w_1, w_2, w_3)$, $\mathsf{L\text{-}input}(g) = w_1$ is the left multiplication input, $\mathsf{R\text{-}input}(g) = w_2$ is the right multiplication input, $\mathsf{D\text{-}input}(g) = w_3$ is the division input, and $\mathsf{coeffs}(g) = a$. The output wire w contains the value $\bar{w} = a\bar{w}_1\bar{w}_2/\bar{w}_3$. Previous works either only considered multiplication gates [BCG+15] or separate multiplication and division gates [BGM17]. Using $\mathsf{multdiv}$ gates can, in some cases, reduce the circuit size compared to separate multiplication and division gates.

Class $\mathcal{C}^{\mathsf{S}}$ contains $\mathbb{F}$-arithmetic circuits $\mathsf{C} : \mathbb{F}^t \to \mathbb{F}^h$, such that:

- For any $w \in \mathsf{inputs}(\mathsf{C})$, there exists $g \in \mathsf{gates}(\mathsf{C})$ such that $\mathsf{type}(g) = \mathsf{multdiv}$, $\overline{\mathsf{inputs}}(g) = (1, \bar{w}, 1)$, and $\mathsf{coeffs}(g) = 1$. That is, each trapdoor itself should be a part of the output of the circuit. Adding those $\mathsf{multdiv}$ gates corresponds to the MPC protocol combining the shares of trapdoor ts of each party to get $[\mathsf{tc}]_z$.
- For any $g \in \mathsf{gates}(\mathsf{C})$:
  - $\mathsf{output}(g) \in \mathsf{outputs}(\mathsf{C})$. Hence, each gate output is a CRS element.
  - If $\mathsf{type}(g) = \mathsf{multdiv}$ then $\mathsf{L\text{-}input}(g) \notin \mathsf{inputs}(C)$, $\mathsf{R\text{-}input}(g), \mathsf{D\text{-}input}(g) \in \mathsf{inputs}(C)$. That is, the left multiplication input can be a constant or an output of a previous gate, the right multiplication and division inputs have to be one of the inputs of the circuit. This allows to easily verify the computation in the MPC. For convenience, we require further that constant value of $\mathsf{L\text{-}input}(g)$ can only be 1; from computational point of view nothing changes since $\mathsf{coeffs}(g)$ can be any constant.
  - If $\mathsf{type}(g) = \mathsf{add}$ then $\mathsf{inputs}(g) \cap \mathsf{inputs}(\mathsf{C}) = \emptyset$. Addition is done locally in MPC (does not require additional rounds) with the outputs of previous gates, since outputs correspond to publicly known CRS elements.

The *sampling depth* $\mathrm{depth}_{\mathsf{S}}$ of a gate $g \in \mathsf{gates}(\mathsf{C})$ is defined as follows:

- $\mathrm{depth}_{\mathsf{S}}(g) = 1$ if g is a $\mathsf{multdiv}$ gate and $\overline{\mathsf{L\text{-}input}}(g)$ is a constant.
- $\mathrm{depth}_{\mathsf{S}}(g) = \max\{\mathrm{depth}_{\mathsf{S}}(g') : g' \text{ an input of } g\}$ for other $\mathsf{multdiv}$ gates,
- $\mathrm{depth}_{\mathsf{S}}(g) = b_g + \max\{\mathrm{depth}_{\mathsf{S}}(g') : g' \text{ an input of } g\}$ for any add gate, where (i) $b_g = 0$ iff all the input gates of g are add gates. (ii) $b_g = 1$, otherwise.

Denote $\mathrm{depth}_{\mathsf{S}}(\mathsf{C}) := \max_g\{\mathrm{depth}_{\mathsf{S}}(g)\}$. We again defined $\mathrm{depth}_{\mathsf{S}}$ slightly differently compared to [BCG+15]; our definition emphasizes the fact that addition gates can be executed locally. Essentially, $\mathsf{N}_p \cdot \mathrm{depth}_{\mathsf{S}}(g_w)$ will be the number of rounds that it takes to compute $[\bar{w}]_z$ with our MPC protocol. The *multiplicative depth* of a circuit (denoted by $\mathrm{depth}_{\mathsf{M}}(\mathsf{C})$) is the maximum number of multiplication gates from any input to any output. An exemplary $\mathcal{C}^{\mathsf{S}}$-circuit is given in 2.3.

Multi-Party Circuit Evaluation Protocol. We describe the circuit evaluation protocol, similar to the one in [BCG+15], that allows to evaluate any $\mathcal{C}^{\mathsf{S}}$-circuits "in the exponent". We assume there are N_p parties $\mathscr{G}_i$, each having published a share $[\mathsf{ts}_{i,s}]_\star \in \mathbb{F}^*$, for $s \in [1\,..\,t]$. The goal of the evaluation protocol is to output

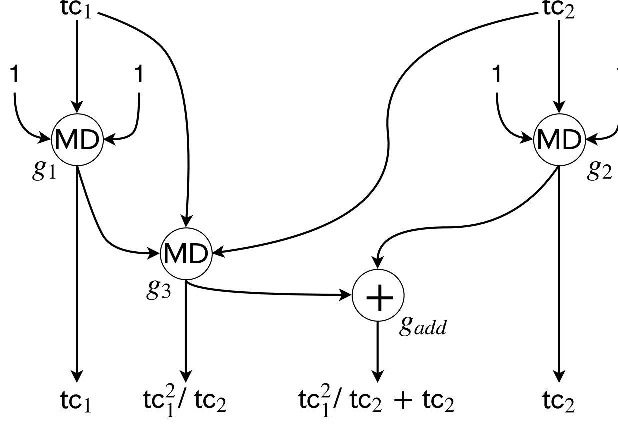


Figure 2.3: Example $\mathcal{C}^{\mathsf{S}}$ circuit with inputs tc_1 and tc_2

$\mathsf{C}_{\mathsf{md}}([b]_\iota, a, \mathsf{ts}_{i,s}, \mathsf{ts}_{i,k})$

return $(a \cdot (\mathsf{ts}_{i,s}/\mathsf{ts}_{i,k}))[b]_\iota$;

$\mathsf{V}_{\mathsf{md}}([b']_\iota, [b]_\iota, a, [\mathsf{ts}_{i,s}]_{3-\iota}, [\mathsf{ts}_{i,k}]_{3-\iota})$

return $(([b']_\iota = [0]_\iota) \vee$
$([b']_\iota \bullet [\mathsf{ts}_{i,k}]_{3-\iota} \neq a[b]_\iota \bullet [\mathsf{ts}_{i,s}]_{3-\iota}))\ ?\ 0 : 1$;

$\mathsf{Eval}_{\mathsf{md}}(a, b, s, k)$

$[b_0]_\iota \leftarrow a[b]_\iota$; // Multiplication with a is done by $\mathcal{G}_1$
for $i \in [1 \mathinner{..} \mathsf{N}_p]$ **do** $\mathcal{G}_i$ broadcasts $[b_i]_\iota \leftarrow \mathsf{C}_{\mathsf{md}}([b_{i-1}]_\iota, 1, \mathsf{ts}_{i,s}, \mathsf{ts}_{i,k})$;
return $[b_{\mathsf{N}_p}]_\iota$;

Figure 2.4: Algorithms C_{md}, V_{md} and the protocol $\mathsf{Eval}_{\mathsf{md}}$ for $\iota \in \{1, 2\}$.

$([\mathsf{C}_1(\mathsf{tc})]_1, [\mathsf{C}_2(\mathsf{tc})]_2)$ where $\mathsf{C}_1, \mathsf{C}_2$ are $\mathcal{C}^{\mathsf{S}}$-circuits and $\mathsf{tc} = (\prod_j \mathsf{ts}_{j,1}, \ldots, \prod_j \mathsf{ts}_{j,t})$. This protocol constructs a well-formed CRS, given that tc is the CRS trapdoor and $[\mathsf{C}_1(\mathsf{tc})]_1$, $[\mathsf{C}_2(\mathsf{tc})]_2$ are respectively all the $\mathbb{G}_1$ and $\mathbb{G}_2$ elements of the CRS. In Section 2.4, we combine the circuit evaluation protocol with a UC-secure commitment scheme to obtain a UC-secure CRS-generation protocol. Each step in the circuit evaluation protocol is publicly verifiable and hence, no trust is needed at all; except that to get the correct distribution we need to trust one party.

We make two significant changes to the circuit evaluation protocol compared to [BCG+15]: (i) we do not require that $\mathsf{C}_1 = \mathsf{C}_2$, allowing CRS elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ to be different, and (ii) instead of multiplication gates we evaluate $\mathsf{multdiv}$ gates.

Let us first describe the computation of $[\bar{w}]_\iota$ for a single gate g_w. For an add gate, given that all input gates have already been computed, that is, $[\bar{w}_1, \ldots, \bar{w}_f]_\iota$ are already public, each $\mathscr{G}_i$ computes $[\bar{w}]_\iota = a_0 + \sum_{j=1}^{f} a_j[\bar{w}_j]_\iota$ locally. A $\mathsf{multdiv}$ gate g, with $\overline{\mathsf{inputs}}(g) = (b, \mathsf{tc}_s, \mathsf{tc}_k)$ and $\mathsf{coeffs}(g) = a$, can be implemented by the N_p-round protocol $\mathsf{Eval}_{\mathsf{md}}$ from 2.4. Here, each party $\mathscr{G}_i$ takes as input $[b]_\iota$ (the output of the preceeding gate or just $[1]_\iota$ if there is none), runs C_{md} procedure on $\mathsf{ts}_{i,s} \in \mathbb{F}$, $\mathsf{ts}_{i,k} \in \mathbb{F}$ (her shares of the trapdoor that are also g's inputs), and broadcasts its output. Note that $[b]_\iota$ corresponds to the left multiplication, $\mathsf{ts}_{i,s}$ to the right multiplication, and $\mathsf{ts}_{i,k}$ to the division input of g.

Importantly, since each party $\mathscr{G}_i$ published $[\{\mathsf{ts}_{i,j}\}_{j=1}^{t}]_\star$, everybody can verify that $\mathscr{G}_i$ executed C_{md} correctly by checking if $\mathsf{V}_{\mathsf{md}}([b_i]_\iota, [b_{i-1}]_\iota, a, [\mathsf{ts}_{i,s}, \mathsf{ts}_{i,k}]_{3-\iota}) = 1$, where $[b_i]_\iota$ is $\mathscr{G}_i$'s output and $[b_{i-1}]_\iota$ is her input (the output of the party $\mathscr{G}_{i-1}$). We assume $[b_0]_\iota = [1]_\iota$ to allow the parties to check the computations of $\mathscr{G}_1$. Just running $\mathsf{Eval}_{\mathsf{md}}$ to evaluate each $\mathsf{multdiv}$ gate in C would require $\approx \mathsf{N}_p \cdot \mathrm{depth}_{\mathsf{M}}(\mathsf{C})$ rounds. Next we see that computation can be parallelized to obtain $\mathsf{N}_p \cdot \mathrm{depth}_{\mathsf{S}}(\mathsf{C})$ rounds.

Optimised Multi-Party Circuit Evaluation Protocol. Before presenting the complete (parallelised) circuit evaluation protocol, we provide an illustrative example of how $\mathcal{C}^{\mathsf{S}}$-circuits can be evaluated efficiently using multiple parties. The idea behind this approach is to allow parties to evaluate the circuit not gate-by-gate but all the gates of the same sampling depth. We say that gates are in the same *layer* if they have the same $\mathrm{depth}_{\mathsf{S}}$. Following the definition of $\mathrm{depth}_{\mathsf{S}}$, layers are separated by add gates. That is, two gates, say g_1 and g_2 are in different layers if there is an add gate g_{add} such that g_1 (or g_2) depends on g_{add}'s output, while the other gate does not. In each layer, each gate is computed using only trapdoor elements and outputs from gates of some preceding layer. Parties evaluate the layer in a round-robin manner broadcasting intermediate values which allows other parties to verify the computation.

This is how the optimised protocol and the naive MPC protocol differ. Since naive protocol evaluates circuit gate-by-gate, one gate's output can be another's input even if both share the same layer. For instance, consider gates g_1 and g_3 from 2.3. There, g_1's output is g_3's input and they are both in the same layer. Since the output of g_1 is computed before g_3 is evaluated, it can be used in the computation. On the other hand, in the optimised version of circuit evaluation all gates in the same layer are evaluated at the same time, thus g_3 is computed at the same time when g_1 is computed.

Example 1. *Suppose we have parties $\mathscr{G}_1$, $\mathscr{G}_2$, $\mathscr{G}_3$ that wish to compute* $\mathsf{crs} = \{[\mathsf{tc}_1]_\star, [\mathsf{tc}_2]_\star, [\mathsf{tc}_1^2/\mathsf{tc}_2]_1, [\mathsf{tc}_1^2/\mathsf{tc}_2 + \mathsf{tc}_2]_1\}$. *Let us only focus on the computation of $\mathbb{G}_1$ elements. This is represented by a $\mathcal{C}^{\mathsf{S}}$-circuit in Fig. 2.3 where we have*

1. *a* multdiv *gate g_1 with input values* $(1, \mathsf{tc}_1, 1)$,
2. *a* multdiv *gate g_2 with input values* $(1, \mathsf{tc}_2, 1)$,
3. *a* multdiv *gate g_3 that takes the output of g_1 as* L-input, *the circuit's inputs* tc_1 *as* R-input, *and* tc_2 *as* D-input, *that is, the input values of g_3 are* $(\mathsf{tc}_1, \mathsf{tc}_1, \mathsf{tc}_2)$, *and*
4. *an add gate g_{add} that adds outputs of g_2 and g_3.*

The parties respectively publish shares $[\mathsf{ts}_{1,1}, \mathsf{ts}_{1,2}]_\star$, $[\mathsf{ts}_{2,1}, \mathsf{ts}_{2,2}]_\star$, $[\mathsf{ts}_{3,1}, \mathsf{ts}_{3,2}]_\star$.

- In the first round, $\mathscr{G}_1$ *broadcasts* $[b^1_{g_1}]_1 \leftarrow [\mathsf{ts}_{1,1}]_1$ *for gate* g_1, $[b^1_{g_2}]_1 \leftarrow [\mathsf{ts}_{1,2}]_1$ *for gate* g_2, *and* $[b^1_{g_3}]_1 \leftarrow [\mathsf{ts}_{1,1}^2/\mathsf{ts}_{1,2}]_1$ *for gate* g_3.
- In the second round, $\mathscr{G}_2$ *broadcasts* $[b^2_{g_1}]_1 \leftarrow \mathsf{ts}_{2,1} \cdot [b^1_{g_1}]_1$ *for gate* g_1, $[b^2_{g_2}]_1 \leftarrow \mathsf{ts}_{2,2} \cdot [b^1_{g_2}]_1$ *for gate* g_2, *and* $[b^2_{g_3,1}]_1 \leftarrow \mathsf{ts}_{2,1} \cdot [b^1_{g_3}]_1$, $[b^2_{g_3,2}]_1 \leftarrow (\mathsf{ts}_{2,1}/\mathsf{ts}_{2,2}) \cdot [b^2_{g_3,1}]_1$ *for* g_3 *(note that g_3 required two computations rather than one).*
- In the third round, $\mathscr{G}_3$ *broadcasts* $[b^3_{g_1}]_1 \leftarrow \mathsf{ts}_{3,1} \cdot [b^2_{g_1}]_1$ *for gate* g_1, $[b^3_{g_2}]_1 \leftarrow \mathsf{ts}_{3,2} \cdot [b^2_{g_2}]_1$ *for gate* g_2, *and* $[b^3_{g_3,1}]_1 \leftarrow \mathsf{ts}_{3,1} \cdot [b^2_{g_3,2}]_1$, $[b^3_{g_3,2}]_1 \leftarrow (\mathsf{ts}_{3,1}/\mathsf{ts}_{3,2}) \cdot [b^3_{g_3,1}]_1$ *for* g_3. *For g_{add} each party computes* $[b_{g_{add}}]_1 \leftarrow [b^3_{g_2}]_1 + [b^3_{g_3,2}]_1$.

Finally, if we define $\mathsf{tc}_1 := \mathsf{ts}_{1,1} \cdot \mathsf{ts}_{2,1} \cdot \mathsf{ts}_{3,1}$ *and* $\mathsf{tc}_2 := \mathsf{ts}_{1,2} \cdot \mathsf{ts}_{2,2} \cdot \mathsf{ts}_{3,2}$, *then the outputs of $\mathscr{G}_3$ contain* $[b^3_{g_1}]_1 = [\mathsf{tc}_1]_1$, $[b^3_{g_2}]_1 = [\mathsf{tc}_2]_1$, *and* $[b^3_{g_3,2}]_1 = [\mathsf{tc}_1^2/\mathsf{tc}_2]_1$*; moreover,* $[b_{g_{add}}]_1 = [\mathsf{tc}_2 + \mathsf{tc}_1^2/\mathsf{tc}_2]_1$. *Besides addition, each element is built up one share multiplication at a time and hence the computation can be verified with pairings, e.g, the last output* $[b^2_{g_3,2}]_1$ *of $\mathscr{G}_2$ is correctly computed exactly when* $[b^2_{g_3,2}]_1 \bullet [\mathsf{ts}_{2,2}]_2 = [b^2_{g_3,1}]_1 \bullet [\mathsf{ts}_{2,1}]_2$. $\square$

Motivated by the example above, we give the full and formal description of the circuit evaluation protocol. Let $\mathsf{C}_\iota \in \mathcal{C}^{\mathsf{S}}$, for $\iota \in \{1, 2\}$, and $\mathsf{C}_{\iota,d} \subseteq \mathsf{gates}(\mathsf{C})$ be a circuit layer that contains all multdiv gates g at sampling depth d. For any $g \in \mathsf{C}_{\iota,d}$ let $\mathsf{ExtractPath}(g, \mathsf{C}_{\iota,d})$ output the longest path $(g_1, \ldots, g_q = g)$ such that each $g_j \in \mathsf{C}_{\iota,d}$, and, for $j < q$, $\mathsf{output}(g_j) = \mathsf{L\text{-}input}(g_{j+1})$. Intuitively, this is the path of gates in $\mathsf{C}_{\iota,d}$ that following only the left inputs lead up to the gate g, say, $\mathsf{ExtractPath}(g_3, \mathsf{C}_{1,1}) = (g_1, g_3)$ for the circuit C in 2.3. For simplicity, we describe a multdiv gate g by a tuple $([b]_\iota, a, s, k)$ where $[b]_\iota = [\overline{\mathsf{L\text{-}input}(g)}]_\iota$ is the left input value, assumed already to be known by the parties, $a = \mathsf{coeffs}(g)$, $\overline{\mathsf{R\text{-}input}}(g) = \mathsf{tc}_s$, and $\overline{\mathsf{D\text{-}input}}(g) = \mathsf{tc}_k$.

The parties evaluate multdiv gates of the circuit in order $\mathsf{C}_{\iota,1}, \mathsf{C}_{\iota,2}, \ldots, \mathsf{C}_{\iota,D_\iota}$, where D_ι is the sampling depth of C_ι. After each layer $C_{\iota,d}$ each party locally evaluates all the addition gates at depth $d+1$. The evaluation of $\mathsf{C}_{\iota,d}$ proceeds in a round-robin fashion. First, $\mathscr{G}_1$ evaluates $\mathsf{C}_{\iota,d}$ with her input shares $\mathsf{ts}_{1,k}$ alone. Next, $\mathscr{G}_2$ multiplies her shares $\mathsf{ts}_{2,k}$ to each output of $\mathscr{G}_1$. However, to make computation verifiable, if $\mathscr{G}_2$ is supposed to compute $[b_g^1 \cdot \mathsf{ts}_{2,\alpha_1} \cdot \ldots \mathsf{ts}_{2,\alpha_q}]_\iota$, where $[b_g^1]_\iota$ is some output of $\mathscr{G}_1$, then it is done one multiplication at a time. Namely, she outputs $[b_{g,1}^2]_\iota = [b_g^1 \cdot \mathsf{ts}_{2,\alpha_1}]_\iota$, $[b_{g,2}^2]_\iota = [b_{g,1}^2 \cdot \mathsf{ts}_{2,\alpha_2}]_\iota$, $\ldots$, $[b_{g,q}^2]_\iota = [b_{g,q-1}^2 \cdot \mathsf{ts}_{2,\alpha_q}]_\iota$. Each multiplication would correspond to exactly one gate in $\mathsf{ExtractPath}(g, \mathsf{C}_{\iota,d})$. The elements $[b_{g,1}^2, \ldots, b_{g,q-1}^2]_\iota$ are used only for verification; $[b_{g,q}^2]_\iota$ is additionally used by $\mathscr{G}_3$ to continue the computation. Each subsequent party $\mathscr{G}_i$ multiplies her shares to the output of $\mathscr{G}_{i-1}$ in a similar fashion. This protocol requires only $\mathsf{N}_p \cdot \mathrm{depth}_{\mathsf{S}}(\mathsf{C}_\iota)$ rounds.

Let $\mathsf{cert}^\iota = (\mathsf{cert}_1^\iota, \ldots, \mathsf{cert}_{D_\iota}^\iota)$ be the total transcript (certificate) in $\mathbb{G}_\iota$ corresponding to the output of the multi-party evaluation of C_ι where cert_r^ι is the transcript in round r. Denote $\mathsf{cert} := (\mathsf{cert}^1, \mathsf{cert}^2)$. All gates of depth r of C_ι are evaluated by a uniquely fixed party $\mathscr{G}_i$. In what follows, let $i = \mathsf{rndplayer}(r)$ be the index of this party.

The complete description of evaluation and verification of a layer $\mathsf{C}_{\iota,d}$ is given in 2.5 with function $\mathsf{C}_{\mathsf{layer}}$ and $\mathsf{V}_{\mathsf{layer}}$ that have the following interface. First, for $i = \mathsf{rndplayer}(r)$ and for both $\iota \in \{1,2\}$, in round r to compute $[\mathsf{C}_{\iota,d}(\mathsf{tc})]_\iota$, $\mathscr{G}_i$ computes $\mathsf{cert}_r^\iota \leftarrow \mathsf{C}_{\mathsf{layer}}(\mathsf{C}_{\iota,d}, \iota, i, r, \{\mathsf{ts}_{i,k}\}_{k=1}^t, \{\mathsf{cert}_j^\iota\}_{j=1}^{r-1})$, given a circuit layer $\mathsf{C}_{\iota,d}$, the shares $\mathsf{ts}_{i,k}$ for all t trapdoors of tc_k, and the transcript $\{\mathsf{cert}_j^\iota\}_{j=1}^{r-1}$ of all previous computation. Second, any party can verify, by using the algorithm $\mathsf{V}_{\mathsf{layer}}(\mathsf{C}_{\iota,d}, \iota, i, r, \{[\mathsf{ts}_{i,k}]_{3-\iota}\}_{k=1}^t, \{\mathsf{cert}_j^\iota\}_{j=1}^r)$, that the computation of the circuit layer $\mathsf{C}_{\iota,d}$ in round r has been performed correctly by $\mathscr{G}_i$. In particular, $\mathscr{G}_i$ checks that $\mathsf{V}_{\mathsf{layer}}$ outputs 1 for all rounds since $\mathscr{G}_i$'s previous round before executing $\mathsf{C}_{\mathsf{layer}}$ for her new round. Importantly, executing $\mathsf{V}_{\mathsf{layer}}$ does not assume the knowledge of any trapdoors.

2.4 UC-Secure CRS Generation

We propose a functionality $\mathcal{F}_{\mathsf{mcrs}}$ for multi-party CRS generation of any $\mathcal{C}^{\mathsf{S}}$-SNARK. Finally, we construct a protocol $\mathsf{K}_{\mathsf{mcrs}}$ that UC-realizes $\mathcal{F}_{\mathsf{mcrs}}$ in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model.

New Ideal Functionality. In Section 2.6, we define the new ideal functionality $\mathcal{F}_{\mathsf{mcrs}} = \mathcal{F}_{\mathsf{mcrs}}^{\mathsf{p},\mathsf{N}_p,\vec{\mathsf{C}},\mathscr{D},\mathsf{comb}}$ for pairing-based (since it outputs elements from $\mathbb{G}_\iota$) multi-party CRS-generation protocol. The CRS is described by a t-input arithmetic circuits $\vec{\mathsf{C}} := (\mathsf{C}_1, \mathsf{C}_2)$ over a field $\mathbb{F} = \mathbb{Z}_p$ such that $\mathsf{crs} = ([\mathsf{C}_1(\mathsf{tc})]_1, [\mathsf{C}_2(\mathsf{tc})]_2)$ for $\mathsf{tc} \leftarrow \mathscr{D}$, where $\mathscr{D}$ is a samplable distribution over $\mathbb{Z}_p^t$.

The trapdoor tc is constructed by combining shares $\mathsf{ts}_i \in \mathtt{Supp}(\mathscr{D})$ of each party $\mathscr{G}_i$ by a function comb. By $\mathtt{Supp}(\mathscr{D})$ we denote the set of all elements in $\mathscr{D}$ that have non-zero probability. For each honest party $\mathscr{G}_i$, the ideal functionality picks $\mathsf{ts}_i \leftarrow \mathscr{D}$, whereas for malicious parties we only know $\mathsf{ts}_i \in \mathtt{Supp}(\mathscr{D})$. The function comb should be defined so that if there exists at least one honest party then $\mathsf{tc} \leftarrow \mathsf{comb}(\mathsf{ts}_1, \ldots, \mathsf{ts}_{\mathsf{N}_p})$ is also distributed accordingly to $\mathscr{D}$. In such case we say that $\mathscr{D}$ is comb-*friendly*. It is true for example when comb is point-wise multiplication and $\mathscr{D}$ is a uniform distribution over $(\mathbb{Z}_p^*)^t$ as, e.g., in [BCG+15, BGG17, BGM17]. This guarantees the correct distribution of crs if at least one party is honest.

We believe $\mathcal{F}_{\mathsf{mcrs}}$ captures essentially any reasonable pairing-based multi-party CRS-generation protocol, where the trapdoor is shared between N_p parties. Note that specifying distinct honest and corrupted inputs to the functionality is common in the UC literature, [BCNP04, KL11]. In 2, we will establish the relation between $\mathcal{F}_{\mathsf{crs}}$ and $\mathcal{F}_{\mathsf{mcrs}}$.

New Protocol. We define the new multi-party CRS-generation protocol $\mathsf{K}_{\mathsf{mcrs}} = \mathsf{K}_{\mathsf{mcrs}}^{\mathsf{p},\mathsf{N}_p,\vec{\mathsf{C}},\mathscr{D},\mathsf{comb}}$ (see 2.7) in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model. This allows us to instantiate the protocol with any DL-extractable commitment and, moreover, the only trust assumption that the protocol needs is the one inherited from the commitment scheme, e.g., using construction from [ABL+19a] gives security in the RPK model. Given that D_ι is the sampling depth

$\mathsf{C}_{\mathsf{layer}}(\mathsf{C}_{\iota,d}, \iota, i, r, \{\mathsf{ts}_{i,k}\}_{k=1}^{t}, \{\mathsf{cert}_j^{\iota}\}_{j=1}^{r-1})$ // The following is executed by $\mathcal{G}_i$

```
 1 :  Assert i = rndplayer(r); cert_r^ι ← ε;
 2 :  for g = ([b]_ι, a, s, k) ∈ C_{ι,d} do // In topological order
 3 :     cert_{g,i} ← ε;    (g_1, ..., g_q) ← ExtractPath(g, C_{ι,d});
 4 :     if i = 1 then
 5 :        if q = 1 then  [b']_ι ← C_md([b]_ι, a, ts_{1,s}, ts_{1,k});
 6 :        else Parse cert_{g_{q-1},1} = [b_L]_ι;
 7 :             [b']_ι ← C_md([b_L]_ι, a, ts_{1,s}, ts_{1,k});
 8 :        cert_{g,1} ← [b']_ι;
 9 :     else Parse cert_{g,i-1} = [b_1, ..., b_q]_ι;  [b']_ι ← [b_q]_ι;
10 :        for j = 1, ..., q do
11 :           Parse g_j = ([b^*]_ι, a^*, s^*, k^*);
12 :           [b']_ι ← C_md([b']_ι, 1, ts_{i,s^*}, ts_{i,k^*});  Append [b']_ι to cert_{g,i};
13 :     Append cert_{g,i} to cert_r^ι;
14 :  return cert_r^ι;
```

$\mathsf{V}_{\mathsf{layer}}(\mathsf{C}_{\iota,d}, \iota, i, r, \{[\mathsf{ts}_{i,k}]_{3-\iota}\}_{k=1}^{t}, \{\mathsf{cert}_j^{\iota}\}_{j=1}^{r})$

```
 1 :  Assert i = rndplayer(r);
 2 :  for each evaluation of [b']_ι ← C_md([b]_ι, a, ts_{i,s}, ts_{i,k}) in round r by G_i do
 3 :     Extract [b']_ι, [b]_ι from {cert_j^ι}_{j=1}^r;
 4 :     if V_md([b']_ι, [b]_ι, a, [ts_{i,s}]_{3-ι}, [ts_{i,k}]_{3-ι}) = 0 then  return 0;
 5 :  return 1;
```

Figure 2.5: $\mathsf{C}_{\mathsf{layer}}$ and $\mathsf{V}_{\mathsf{layer}}$ for $\iota \in \{1, 2\}$

Parameters: p defines a bilinear pairing, $\vec{\mathsf{C}} = (\mathsf{C}_1, \mathsf{C}_2)$ contains t-input arithmetic circuits over the field $\mathbb{Z}_p$, $\mathscr{D}$ is a distribution of trapdoor elements, and $\mathsf{comb} : (\mathbb{Z}_p^t)^{\mathsf{N}_p} \rightarrow \mathbb{Z}_p^t$. We have parties $\mathscr{G}_i$ for $i \in [1 \mathinner{..} \mathsf{N}_p]$.

Share collection phase:

- Upon receiving $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i)$ from an honest $\mathscr{G}_i$, store $\mathsf{ts}_i \leftarrow \mathscr{D}$ and send $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i)$ to Sim.
- Upon receiving $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i, \mathsf{ts}_i)$ from a dishonest $\mathscr{G}_i$, if $\mathsf{ts}_i \in \mathsf{supp}(\mathscr{D})$, then store ts_i, else abort.

Only one message from each $\mathscr{G}_i$ is accepted.

CRS generation phase: Once ts_i is stored for each $\mathscr{G}_i$:

- Compute $\mathsf{tc} \leftarrow \mathsf{comb}(\mathsf{ts}_1, \ldots, \mathsf{ts}_{\mathsf{N}_p})$.
- Set $\mathsf{crs} \leftarrow ([\mathsf{C}_1(\mathsf{tc})]_1, [\mathsf{C}_2(\mathsf{tc})]_2)$ and send $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$ to Sim.
- If Sim returns $(\mathtt{CRS}, \mathsf{ok})$ then send $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$ to every party $\mathscr{G}_i$ for $i \in [1 \mathinner{..} \mathsf{N}_p]$.

Figure 2.6: Ideal functionality $\mathcal{F}_{\mathsf{mcrs}}$

of C_ι, then $R = \mathsf{N}_p \cdot \max(D_1, D_2)$ is the number of rounds needed to evaluate both circuits in parallel. For the sake of simplicity, we assume cert_r^ι is the empty string for $r > \mathsf{N}_p \cdot D_\iota$.

$\mathsf{K}_{\mathsf{mcrs}}$ proceeds in rounds: (i) In round 1, each $\mathscr{G}_i$ gets a signal $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i)$; parties commit to their shares of trapdoor tc. (ii) In round 2, each party $\mathscr{G}_i$ gets a signal $(\mathtt{mcrsopen}, \mathsf{sid})$; parties open their shares. (iii) In round $r \in [3 \mathinner{..} R+2]$, $(\mathtt{mcrscertok}, \mathsf{sid}, \mathscr{G}_i, r)$ is triggered, where $i = \mathsf{rndplayer}(r)$; parties jointly compute crs from the trapdoor shares; before party $\mathscr{G}_i$ performs her computation, she checks if previous computation were done correctly. (iv) In round $R+3$, each party $\mathscr{G}_i$ gets the signal $(\mathtt{mcrsfinal}, \mathsf{sid}, \mathscr{G}_i)$ and extracts the crs from cert. The CRS will be output by $\mathscr{G}_i$ only if all the verifications succeeded.

The signals $\mathtt{sc}$, $\mathtt{mcrsopen}$, $\mathtt{mcrscertok}$, and $\mathtt{mcrsfinal}$ can be sent either by a controller server or by the internal clock of $\mathscr{G}_i$. The construction uses a secure broadcast channel; thus, if a message is broadcast, then all parties are guaranteed to receive the same message. Note that after $\mathscr{G}_j$ obtains $(\mathtt{rcpt}, \mathsf{lbl}_{ijk})$, for $i \in [1 \mathinner{..} \mathsf{N}_p], j \neq i, k \in [1 \mathinner{..} t]$, she broadcasts $(\mathtt{mcrsreceipt}, \mathsf{lbl}_{ijk})$ since $\mathtt{rcpt}$ is not broadcast.

Security. To prove UC-security of $\mathsf{K}_{\mathsf{mcrs}}$, we restrict $\mathcal{F}_{\mathsf{mcrs}}$ as follows: (i) $\vec{\mathsf{C}} = (\mathsf{C}_1, \mathsf{C}_2)$ such that $\mathsf{C}_\iota \in \mathcal{C}^{\mathsf{S}}$ for $\iota \in \{1, 2\}$. Note that this means that for any trapdoor element $\mathsf{tc}_k \in \mathsf{tc}$, $[\mathsf{tc}_k]_\star \in \mathsf{crs}$. (ii) $\mathscr{D}$ is the uniform distribution on $(\mathbb{Z}_p^*)^t$, (iii) $\mathsf{comb}(\mathsf{ts}_1, \ldots, \mathsf{ts}_{\mathsf{N}_p}) := \mathsf{ts}_1 \circ \ldots \circ \mathsf{ts}_{\mathsf{N}_p}$, where $\circ$ denotes point-wise multiplication, and ts_{ik} is $\mathscr{G}_i$'s share of tc_k.

Theorem 1. $\mathsf{K}_{\mathsf{mcrs}}$ *UC-realizes* $\mathcal{F}_{\mathsf{mcrs}}$ *in the* $\mathcal{F}_{\mathsf{dlmcom}}$*-hybrid model with perfect security against a static adversary. Formally, there exits a PPT simulator* $\mathsf{Sim}^{\mathcal{A}}$ *such that for every static (covert) PPT adversary* $\mathcal{A}$ *and for any non-uniform PPT environment* $\mathcal{Z}$*,* $\mathcal{Z}$ *cannot distinguish* $\mathsf{K}_{\mathsf{mcrs}}$ *composed with* $\mathcal{F}_{\mathsf{dlmcom}}$ *and* $\mathcal{A}$ *from* Sim *composed with* $\mathcal{F}_{\mathsf{mcrs}}$*. That is,* $\mathsf{HYBRID}^{\mathcal{F}_{\mathsf{dlmcom}}}_{\mathsf{K}_{\mathsf{mcrs}}, \mathcal{A}, \mathcal{Z}} = \mathsf{IDEAL}_{\mathcal{F}_{\mathsf{mcrs}}, \mathsf{Sim}^{\mathcal{A}}, \mathcal{Z}}$.

Sketch. To prove UC-security, we have to construct an algorithm Sim that is able to simulate behaviour of honest parties for $\mathcal{Z}$ in the ideal world without knowing their real inputs. Since we are in $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model, Sim simulates $\mathcal{F}_{\mathsf{dlmcom}}$ for the malicious parties and hence learns their shares. At first, Sim picks random shares ts'_{ik} to simulate all the honest parties. Once the ideal functionality has received $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i)$ from all the honest parties and $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i, \mathsf{ts}_i)$ for all the dishonest parties (forwarded by Sim), then Sim receives $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$ from the ideal functionality. Now, Sim fixes one honest party $\mathscr{G}_h$ and opens its commitments to $[\mathsf{ts}^*_{hk}]_1 \leftarrow [\mathsf{tc}_k]_1 / (\prod_{i \in [1 \mathinner{..} \mathsf{N}_p] \setminus \{h\}} \mathsf{ts}'_{ik})$ for $[\mathsf{tc}_k]_1 \in \mathsf{crs}$ and ts'_{ik} collected through $\mathcal{F}_{\mathsf{dlmcom}}$. Since $[\mathsf{tc}_k]_1$ is uniformly random, then so is $[\mathsf{ts}^*_{hk}]_1$ (the same distibution as in the real protocol). With a similar strategy Sim simulates each gate output for $\mathscr{G}_h$ such that the final output of the simulated protocol is crs, matching the output of the ideal functionality. □

2.5 Secure MPC for NIZKs

Next, we show that $\mathsf{K}_{\mathsf{mcrs}}$ can be used to generate the CRS of any $\mathcal{C}^{\mathsf{S}}$-SNARK without harming the completeness, soundness, or (subversion) zero-knowledge properties. It could also be used to generate CRS of other primitives which can be represented by $\mathcal{C}^{\mathsf{S}}$-circuits, but it is especially well suited for the intricate structure of SNARK CRS. Finally, we apply the protocol to the Sub-ZK secure version [ABLZ17, Fuc18] of the most efficient zk-SNARK by Groth [Gro16].

NIZK in the MCRS model. Let Ψ be a NIZK argument system secure in the $\mathcal{F}_{\mathsf{crs}}$-hybrid model. We show that by instantiating $\mathcal{F}_{\mathsf{crs}}$ with $\mathcal{F}_{\mathsf{mcrs}}$, the NIZK remains complete, sound, and zero-knowledge, provided that the adversary $\mathcal{A}$ controls up to $\mathsf{N}_p - 1$ out of N_p parties. Here we require that $\mathscr{D}$ is comb-friendly. See Fig. 2.8 for the high-level description of MPC protocol for the CRS generation.

Theorem 2. *Let* $\mathscr{D}$ *and* $\mathsf{comb} : (\mathsf{Supp}(\mathscr{D}))^{\mathsf{N}_p} \rightarrow \mathsf{Supp}(\mathscr{D})$ *be such that* $\mathscr{D}$ *is* comb*-friendly.* $\mathsf{K}^{\mathcal{F}_{\mathsf{mcrs}}}_{\mathsf{crs}}$ *securely realizes* $\mathcal{F}^{\mathscr{D}, f}_{\mathsf{crs}}$ *in the* $\mathcal{F}_{\mathsf{mcrs}}$*-hybrid model given (covert)* $\mathcal{A}$ *corrupts up to* $\mathsf{N}_p - 1$ *out of* N_p *parties (i.e. CRS generators).*

Share collection phase: *Round 1:* upon receiving $(\mathtt{sc}, \mathsf{sid}, \mathscr{G}_i)$, $\mathscr{G}_i$ does the following.
for $k \in [1\,..\,t]$ **do**

1. $\mathsf{ts}_{ik} \leftarrow \mathbb{Z}_p^*$;
2. **for** $j \neq i$ **do**
   - Send $(\mathtt{commit}, \mathsf{sid}, \mathsf{cid}_{ijk}, \mathscr{G}_i, \mathscr{G}_j, \mathsf{ts}_{ik})$ to $\mathcal{F}_{\mathsf{dlmcom}}$;
   - Upon receiving $(\mathtt{rcpt}, \mathsf{lbl}_{ijk} = (\mathsf{sid}, \mathsf{cid}_{ijk}, \mathscr{G}_i, \mathscr{G}_j))$, $\mathscr{G}_j$ broadcasts $(\mathtt{mcrsreceipt}, \mathsf{lbl}_{ijk})$;
   - Store $\mathsf{st}_{ij} \leftarrow (\mathsf{lbl}_{ijk}, \mathsf{ts}_{ik})_{k=1}^t$;

If by the end of the round 1, $\mathscr{G}_i$ does not receive $(\mathtt{mcrsreceipt}, \mathsf{sid}, \mathsf{cid}_{jj'k}, \mathscr{G}_j, \mathscr{G}_{j'})$ for $k \in [1\,..\,t]$, $j \neq i$, $j' \neq i$, and $j' \neq j$ then $\mathscr{G}_i$ aborts.

Round 2: upon receiving $(\mathtt{mcrsopen}, \mathsf{sid})$, $\mathscr{G}_i$ does:
for $k \in [1\,..\,t]$ **do**

1. **for** $j \neq i$ **do**
   - Send $(\mathtt{open}, \mathsf{sid}, \mathsf{cid}_{ijk})$ to $\mathcal{F}_{\mathsf{dlmcom}}$;
   - After receiving $(\mathtt{open}, \mathsf{lbl}_{ijk}, [\mathsf{ts}'_{ijk}]_1)$, where $\mathsf{lbl}_{ijk} = (\mathsf{sid}, \mathsf{cid}_{ijk}, \mathscr{G}_i, \mathscr{G}_j)$, from $\mathcal{F}_{\mathsf{dlmcom}}$, $\mathscr{G}_j$ stores $(\mathsf{lbl}_{ijk}, [\mathsf{ts}'_{ijk}]_1)$; // Comment: If $\mathscr{G}_i$ is honest then $\mathsf{ts}_{ik} = \mathsf{ts}'_{ijk}$
2. Broadcast $(\mathtt{sbroadc}, \mathscr{G}_i, k, [\mathsf{ts}_{ik}]_1)$.
3. Upon receiving $(\mathtt{sbroadc}, \mathscr{G}_i, k, [\mathsf{ts}_{ik}]_1)$ broadcast by $\mathscr{G}_i$, $\mathscr{G}_j$ does the following.
   - If $(\mathsf{lbl}_{ijk}, [\mathsf{ts}'_{ijk}]_1)$ is not stored for some $[\mathsf{ts}'_{ijk}]_1$ then abort.
   - Abort unless $[\mathsf{ts}_{ik}]_1 = [\mathsf{ts}'_{ijk}]_1 \neq [0]_1$.
   - If by the end of round 2, $\mathscr{G}_j$ has not received $(\mathtt{sbroadc}, \dots)$, $\forall j \neq i$, $\forall k$, then $\mathscr{G}_j$ aborts.

CRS generation phase: *Round $r = 3$ to $R + 2$:*
upon receiving $(\mathtt{mcrscertok}, \mathsf{sid}, \mathscr{G}_i, r)$, $\mathscr{G}_i$ does the following, for $i = \mathsf{rndplayer}(r)$.

1. Extract $\mathsf{C}_{1,d}$, $\mathsf{C}_{2,d}$ corresponding to round r from C_1, C_2;
2. **for** $\iota \in \{1, 2\}$ **do** $\mathsf{cert}_r^\iota \leftarrow \mathsf{C}_{\mathsf{layer}}(\mathsf{C}_{\iota,d}, \iota, i, r, \{\mathsf{ts}_{i,k}\}_{k=1}^t, \{\mathsf{cert}_j^\iota\}_{j=1}^{r-1})$;
3. $\mathsf{cert}_r \leftarrow (\mathsf{cert}_r^1, \mathsf{cert}_r^2)$; broadcast $(\mathtt{mcrscert}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, r, \mathsf{cert}_r)$;
4. Any $j \neq i$ does after receiving $(\mathtt{mcrscert}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, r, \mathsf{cert}_r)$ from $\mathscr{G}_i$:
   - if $j \neq \mathsf{rndplayer}(r)$, $\mathsf{V}_{\mathsf{layer}}(\mathsf{C}_{1,d}, \iota, i, r, \{[\mathsf{ts}_{i,k}]_{3-\iota}\}_{k=1}^t, \{\mathsf{cert}_k^1\}_{k=1}^r)) = 0$, or $\mathsf{V}_{\mathsf{layer}}(\mathsf{C}_{2,d}, \iota, i, r, \{[\mathsf{ts}_{i,k}]_{3-\iota}\}_{k=1}^t, \{\mathsf{cert}_j^2\}_{j=1}^r)) = 0$ then abort;
   - Replace stored $(\mathsf{sid}, \mathsf{cid}, r-1, \{\mathsf{cert}_j^\iota\}_{j=1}^{r-1})$ with $(\mathsf{sid}, \mathsf{cid}, r, \{\mathsf{cert}_j^\iota\}_{j=1}^r)$;

If by the end of round r, for any i, $\mathscr{G}_i$ has not stored $(\mathtt{mcrscert}, \mathsf{sid}, \mathsf{cid}, \mathscr{G}_i, r, \mathsf{cert}_r)$ then $\mathscr{G}_i$ aborts.

Round $R + 3$: upon receiving $(\mathtt{mcrsfinal}, \mathsf{sid}, \mathscr{G}_i)$, $\mathscr{G}_i$ does the following.

1. If $\mathscr{G}_i$ has already received this message then ignore;
2. Extract crs from $\{\mathsf{cert}_k^1, \mathsf{cert}_k^2\}_{k=1}^R$. Write $(\mathtt{CRS}, \mathsf{crs})$ on the output tape.

Figure 2.7: The protocol $\mathsf{K}_{\mathsf{mcrs}}$ in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model

$\mathsf{K}_{\mathsf{crs}}^{\mathcal{F}_{\mathsf{mcrs}}}$ proceeds as follows, running with a set $\{P_1, \ldots, P_{\mathsf{N}'_p}\}$ of parties, designated set $\{\mathscr{G}_1, \ldots, \mathscr{G}_{\mathsf{N}_p}\}$ of CRS generators, and an adversary Sim.
CRS generation: Send a signal to each $\mathscr{G}_i$ to execute the functionality $\mathcal{F}_{\mathsf{mcrs}}$. If $\mathcal{F}_{\mathsf{mcrs}}$ returns crs then $\mathscr{G}_i$ stores $(\mathsf{sid}, \mathsf{crs})$.
Retrieval: P_i sends $(\mathtt{retrieve}, \mathsf{sid})$ to each $\mathscr{G}_j$: If $(\mathsf{sid}, \mathsf{crs})$ is recorded for some crs then $\mathscr{G}_j$ sends $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$. If all N_p responses from $\mathscr{G}_j$ are the same, then P_i outputs $(\mathtt{CRS}, \mathsf{sid}, \mathsf{crs})$. Else P_i aborts.

Figure 2.8: Protocol $\mathsf{K}_{\mathsf{crs}}^{\mathcal{F}_{\mathsf{mcrs}}}$

CRS / trapdoor: $\mathsf{tc} \leftarrow (\alpha, \beta, \gamma, \delta, \chi)$ and $\mathsf{crs} = (\mathsf{crs_P}, \mathsf{crs_V}, \mathsf{crs_{CV}})$, where

$$\mathsf{crs_P} \leftarrow \left(\begin{array}{l} \left[\alpha, \beta, \delta, \left(\left(u_j(\chi)\beta + v_j(\chi)\alpha + w_j(\chi) \right) / \delta \right)_{j=m_0+1}^{\mathsf{m}} \right]_1, \\ \left[\left(\chi^i \ell(\chi)/\delta \right)_{i=0}^{\mathsf{n}-2}, \left(u_j(\chi), v_j(\chi) \right)_{j=0}^{\mathsf{m}} \right]_1, \left[\beta, \delta, \left(v_j(\chi) \right)_{j=0}^{\mathsf{m}} \right]_2 \end{array} \right),$$

$$\mathsf{crs_V} \leftarrow \left(\left[\left(\left(u_j(\chi)\beta + v_j(\chi)\alpha + w_j(\chi) \right) / \gamma \right)_{j=0}^{m_0} \right]_1, [\gamma, \delta]_2 \right),$$

$$\mathsf{crs_{CV}} \leftarrow ([\gamma, (\chi^i)_{i=1}^{\mathsf{n}-1}, (\ell_i(\chi))_{i=1}^{\mathsf{n}}]_1, [\alpha, \chi, \chi^{\mathsf{n}-1}]_2).$$

Figure 2.9: CRS of Z^* Sub-ZK SNARK from [ABLZ17]

The proof of this theorem is given in the full version of the paper. Next corollary immediately follows from the universal composition theorem [Can01].

Corollary 1. *Let Ψ be a NIZK argument that is complete, sound, computationally ZK, and computationally Sub-ZK in the $\mathcal{F}_{\mathsf{crs}}^{\mathscr{D},f}$-hybrid model. By instantiating $\mathcal{F}_{\mathsf{crs}}^{\mathscr{D},f}$ with $\mathsf{K}_{\mathsf{crs}}^{\mathcal{F}_{\mathsf{mcrs}}}$, the following holds:*

- *Ψ is complete, sound, and computationally zero-knowledge in the $\mathcal{F}_{\mathsf{mcrs}}$-hybrid model, given that (covert) $\mathcal{A}$ corrupts up to $\mathsf{N}_p - 1$ out of N_p parties.*
- *Ψ is Sub-ZK in the $\mathcal{F}_{\mathsf{mcrs}}$-hybrid model, even if (covert) $\mathcal{A}$ corrupts all N_p parties.*
- *If $\mathscr{D}$ is a uniform distribution over $(\mathbb{Z}_p^*)^t$, comb the point-wise multiplication and the CRS can be computed by $\mathcal{C}^{\mathsf{S}}$-circuits, then properties 1 and 1 hold in the $\mathcal{F}_{\mathsf{dlmcom}}$-hybrid model since $\mathsf{K}_{\mathsf{mcrs}}$ realizes $\mathcal{F}_{\mathsf{mcrs}}$ in that setting.*

Applying $\mathsf{K}_{\mathsf{mcrs}}$ to Groth's zk-SNARK. Fig. 2.9 contains the description of the CRS for the Sub-ZK version of Groth's zk-SNARK Z^* as was proposed in [ABLZ17]. We have omitted the element $[\alpha\beta]_T$ that can be computed from $[\alpha]_1$ and $[\beta]_2$. The CRS from [ABLZ17] differs from the original CRS for Groth's zk-SNARK [Gro16] by the entries in $\mathsf{crs_{CV}}$ which make the CRS verifiable using a CV algorithm. Here, $\ell_i(X)$ are Lagrange basis polynomials and $\ell(X) = X^{\mathsf{n}} - 1$, $u_j(X)$, $v_j(X)$, $w_j(X)$ are publicly-known circuit-dependent polynomials. Due to the lack of space, we do not present other algorithms of Z^*.

We recall that to use the algorithm $\mathsf{K}_{\mathsf{crs}}^{\mathcal{F}_{\mathsf{mcrs}}}$ the CRS has to be of the form $\mathsf{crs} = ([\mathsf{C}_1(\mathsf{tc})]_1, [\mathsf{C}_2(\mathsf{tc})]_2)$, where $\mathsf{C}_\iota \in \mathcal{C}^{\mathsf{S}}$. In Fig. 2.9, the highlighted entries cannot be computed from trapdoors by a $\mathcal{C}^{\mathsf{S}}$-circuit unless we add $\mathsf{crs_{TV}} = ([(w_j(\chi), \beta u_j(\chi), \alpha v_j(\chi))_{j=0}^{\mathsf{m}}, \chi^{\mathsf{n}}]_1, [(\ell_i(\chi))_{i=1}^{\mathsf{n}}, (\chi^k)_{k=1}^{\mathsf{n}-1}]_2)$ to the CRS. To obtain better efficiency we additionally add $[(\ell_i(\chi))_{i=1}^{\mathsf{n}}]_2$ to the CRS, although they can be computed from the existing elements $[(\chi^k)_{k=1}^{\mathsf{n}-1}]_2$.

However, since we are adding elements to the CRS, we also need to reprove the soundness. We do this in the full version of the paper.

We give a brief description of the CRS-generation protocol for Z^* without explicitly describing the circuits C_1 and C_2. Without directly saying it, it is assumed that parties verify all the computations as shown in Fig. 2.7.

Share collection phase. Parties proceed as is in Fig. 2.7 to produce random and independent shares $[\mathsf{ts}_i]_\star = [\alpha_i, \beta_i, \gamma_i, \delta_i, \chi_i]_\star$ for each $\mathscr{G}_i$.

CRS generation phase. (i) On layers $\mathsf{C}_{1,1}, \mathsf{C}_{2,1}$ parties jointly compute $[\alpha, \beta, \gamma, \delta]_\star$, $[(\chi^k)_{k=1}^{\mathsf{n}-1}]_\star$ and $[\chi^{\mathsf{n}}]_1$. (ii) Each $\mathscr{G}_i$ locally computes $[(\ell_k(\chi))_{k=1}^{\mathsf{n}}]_\star$, $[(w_j(\chi), u_j(\chi))_{j=0}^{\mathsf{m}}]_1$, and $[(v_j(\chi))_{j=0}^{\mathsf{m}}]_\star$ using $[(\chi^k)_{k=1}^{\mathsf{n}-1}]_\star$; and also computes $[\ell(\chi)]_1 = [\chi^{\mathsf{n}}]_1 - [1]_1$. (iii) On layer $\mathsf{C}_{1,2}$, from input $[\ell(\chi)]_1$, parties jointly compute $[(\chi^k \ell(\chi)/\delta)_{k=0}^{\mathsf{n}-2}]_1$ using $\mathsf{n} - 1$ $\mathsf{multdiv}$ gates. Moreover, they compute $[(\beta u_l(\chi), \alpha v_l(\chi))_{l=0}^{\mathsf{m}}]_1$. (iv) Each party computes locally $[(\beta u_l(\chi) + \alpha v_l(\chi) + w_l(\chi))_{l=0}^{\mathsf{m}}]_1$. (v) On layer $\mathsf{C}_{1,3}$ parties compute jointly $[(\beta u_l(\chi) + \alpha v_l(\chi) + w_l(\chi)/\gamma)_{l=0}^{m_0}]_1$ and $[(\beta u_l(\chi) + \alpha v_l(\chi) + w_l(\chi)/\delta)_{l=m_0+1}^{\mathsf{m}}]_1$.

The cost of the CRS generation for Z^* can be summarised as follows: the circuits C_1 and C_2 have both sampling depth 3; the multi-party protocol for computing the crs takes $3\mathsf{N}_p + 6$ rounds and requires $3\mathsf{m} + 3\mathsf{n} + 9$ $\mathsf{multdiv}$ gates. Note that with separate multiplication and division gates one would need $2\mathsf{m} + 3\mathsf{n} + 8$ multiplication gates and $\mathsf{m} + \mathsf{n}$ division gates which would be less efficient.

Chapter 3

On the Efficiency of Privacy-Preserving Smart Contract Systems

3.1 Introduction

Eliminating the need for a trusted third party in monetary transactions, consequently enabling direct transactions between individuals is one of the main achievements in the cryptocurrencies such as Bitcoin. Importantly, it is shown that the technology behind cryptocurrencies has more potential than what is only used in direct transactions. Different blockchain-based systems such as smart contracts [KMS+15, JKS16], distributed cloud storages [WLB14], digital coins such as Ethereum [Woo14] are evidence of why blockchain technology offers much more functionalities than what we can see in Bitcoin. Smart contracts belong to the popular applications of blockchain technology that recently saw increased interest. A smart contract is a generic term denoting programs written in cryptocurrency scripting languages, that involves digital assets and some parties. The parties deposit assets into the contract and the contract redistributes the assets among the parties based on provisions of the smart contract and inputs of the parties.

Different research has shown that even if payments (e.g. in Bitcoin) or interconnections (e.g. in smart contracts) are conducted between pseudorandom addresses, privacy of end-users is lacking. Indeed, this mostly arises from the nature of technology that a decentralized publicly shared ledger records list of transactions along with related information (e.g. addresses of parties, transferred values, etc), and long-time monitoring and some data analysis (e.g. transaction graph analysis) on this ledger usually reveals some information about the identity of end-users. To address these concerns and provide strong privacy for end-users, several alternatives to Bitcoin protocol and smart contract systems have been proposed; e.g. confidential assets [PBF+18], privacy-preserving auditing [NVV18], privacy-preserving cryptocurrencies such as Zerocash [BCG+14] and Monero [Noe15], privacy-preserving smart contract systems such as Hawk [KMS+15] and Gyges [JKS16].

Zerocash and Monero are two known anonymous cryptocurrencies that provide privacy for end-users. Each of them uses different cryptographic tools to guarantee strong privacy. Monero uses ring signatures that cryptoeprint:2015:675 an individual from a group to provide a signature such that it is impossible to identify which member of that group made the signature. On the other side, Zerocash uses zero-knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs [Gro10, Lip12, PHGR13, BCTV13, Gro16, GM17]) to prove the correctness of all computations inside a direct transaction, without revealing the source, destination and values of the transferred coins. In a similar technique, privacy-preserving smart contract system Hawk [KMS+15] and criminal smart contract system Gyges [JKS16] use universally composable zk-SNARKs to provide anonymous interconnection and payment.

zk-SNARKs. Among various Non-Interactive Zero-Knowledge (NIZK) arguments, zk-SNARKs are one of the most popular ones in practical systems. This is mainly because of their succinct proofs, and consequently very efficient verification. A zk-SNARK proof allows one to efficiently verify the veracity of statements without learn-

ing extra information about the prover. The proofs can be verified offline very quickly (in a few milliseconds) by possibly many independent verifiers. This can be very effective in efficiency of large-scale distributed systems. Efficiency of zk-SNARKs mainly comes from the fact that their construction relies on non-falsifiable assumptions (e.g. knowledge assumptions [Dam]) that allow succinct proofs and non-black-box extraction in security proofs. On the other hand, a zk-SNARK with non-black-box extraction cannot achieve Universally Composable Security (UC-security) which is imperative and necessary in constructing larger cryptographic systems [Can01]. Du to this fact, zk-SNAKRs cannot be directly adopted in larger systems that should guarantee UC-security.

Privacy-preserving smart contract systems. Recently, some elegant UC-based frameworks are presented that allow to construct privacy-preserving smart contracts, including Hawk [KMS+15] and Gyges [JKS16] for criminal smart contracts. These systems record zk-SNARK proofs on ledger, instead of public transactions between pseudonyms, which brings stronger transactional privacy. Strictly speaking, Hawk is a system that gets a program and compiles it to a cryptographic protocol between the contract correspondents (including users and a manager) and the blockchain. It consists of two main blocks, where one is responsible for *private money transfers* and uses a variation of Zerocash [BCG+14], while the second part handles other contract-defined operations of the system. Similar to Zerocash, operations such as *Mint*, which is required in minting a new coin, and *Pour*, which enables anonymous transactions, are located in the first block. On the other side, contract-related operations such as *Freeze*, *Compute* and *Finalize*, that are three necessary operations defined by Hawk for each smart contract, are addressed in the second block. More details regard to the mentioned operations can be found in [KMS+15] [1]. To achieve anonymity in the mentioned operations and payments, Hawk widely uses zk-SNARKs to prove different statements. As the whole system intended to achieve UC-security, so they needed to use a UC-secure zk-SNARK in the system. Additionally, since Zerocash also uses a non-UC-secure zk-SNARK and it does not satisfy UC-security, so to make it useable in Hawk, they needed a variation of Zerocash that uses a UC-secure zk-SNARK and also guarantees UC-security. To this aim, designers of Hawk have used COCO framework [KZM+15a] (a framework to lift a non-UC-secure sound NIZK to a UC-secure one; COCO stands for *Composable 0-knowledge, Compact 0-knowledge*; a formal description of the framework is provided in section 3.2.3) to lift the non-UC-secure zk-SNARK used in Zerocash [BCTV13], to a UC-secure zk-SNARK, such that the lifted scheme can be securely used in composition with the rest of system [Can01]. Then, due to using a UC-secure zk-SNARK in Zerocash, designer of Hawk modified the structure of original Zerocash and used the customized version in their system, which also guarantees UC-security. The lifted UC-secure zk-SNARK frequently is used in the system and plays an essential role in the efficiency of entire system.

In the performance evaluation of Hawk [KMS+15] authors show that the efficiency of their system severely depend on efficiency of the lifted UC-secure zk-SNARK (which is the case in Gyges [JKS16] as well). In fact, computational complexity of both systems are dominated with complexity of the underlying UC-secure zk-SNARK. Particularly, Kosba et al. [KMS+15] emphasize that practical efficiency is a permanent goal of Hawk's design, so to get the best, they also propose various optimizations. By considering this, one may ask, can we improve efficiency of the underlying UC-secure zk-SNARKs such that the efficiency of complete systems will be improved?

Our contribution. We show that one can improve efficiency of Hawk (and similarly Gyges) smart contract system by improving the efficiency of underlying UC-secure zk-SNARK. We will see that one can use a similar approach used by Kosba et al. (in Hawk [KMS+15]) and Juels et al. (in Gyges [JKS16]) and construct a UC-secure version of Groth and Maller's zk-SNARK [GM17] (refereed as GM zk-SNARK in the rest), that has simpler construction and better efficiency than the ones that currently are used in the systems. To do so, we slightly modify the construction of GM zk-SNARK by enforcing the prover to send encryption of witnesses along with the proof, and then show that it achieves black-box simulation extractability, equivalently UC-security, which allows to deploy in both systems to improve cryptoeprint:2015:675.

[1] A tutorial about the system can be found in http://cryptowiki.net/index.php?title=Privacy_preserving_smart_contracts:_Hawk_project

Both Hawk and Gyges have used COCO framework to lift a variation of Pinocchio zk-SNARK [PHGR13] which is deployed in Zerocash (proposed by Ben Sasson et al. [BCTV13]). Later it details we show that, as GM zk-SNARK [GM17] has better efficiency than the mentioned variation of Pinocchio zk-SNARK, and as our changes are lighter than the changes that are applied on Ben Sasson et al.'s zk-SNARK in Hawk [KMS+15] and Gyges [JKS16], so we get a UC-secure zk-SNARK that has simpler construction and better efficiency than the ones that currently are deployed in the systems. Indeed, we will see that our changes are a small part of their changes, which leads to have less overload.

In the modified construction, we do the changes in CRS circuit level and try to keep the prover and verifier procedure as original one that both are considerably optimized in the original construction [GM17]. We believe, new constructed UC-secure zk-SNARK can be of independent interest and it can be deployed in any large cryptographic system that aims to guarantee UC-security and needs to use zk-SNARKs.

Discussion of UC-secure NIZKs. Most of efficient zk-SNARKs only guarantee *knowledge soundness*, meaning that if an adversary can come up with a valid proof, there exists an extractor that can extract the witness from the adversary. But in some cases, e.g. in signatures of knowledge SoKs [CL06], *knowledge soundness* is not enough, and one needs more security guarantee. More accurately, most of zk-SNARKs are vulnerable to the malleability attack which allows an adversary to modify an old proof to a new valid one, that is not desired in some cases. To address this, the notion of *simulation exractability* is defined which ensures that an adversary cannot come up with a new acceptable proof (or an argument), even if he already has seen arbitrary simulated proofs, unless he knows the witness. In other words, simulation extractability implies that if an adversary, who has obtained arbitrary number of simulated proofs, can generate an acceptable new proof for a statement, there exists an extractor that can extract the witness. Based on extraction procedure which is categorized as Black-Box (BB) or non-Black-Box (nBB), there are various notions of simulation extractibility [Gro06, KZM+15a, GM17]. In BB extraction, there exists a black-box (universal) extractor which can extract the witness from all adversaries, however in the nBB extraction, for each adversary there exists a particular extractor that can extract only if it has access to the adversary's source code and random coins. It is already observed and proven that a NIZK system that achieves simulation extractibility with BB extraction, can guarantee the UC-security [CLOS02, Gro06, GOS06].Therefore, constructing a simulation-extractable zk-SNARK with BB extraction is equivalent to constructing a UC-secure zk-SNARK (which the proof will be only circuit succinct). Strictly speaking, in a UC-secure NIZK the simulator of *ideal-world* should be able to extract witnesses without getting access to the source code of environment's algorithm, which this is guaranteed by BB extraction.

A known technique to achieve a simulation-extractable NIZK with BB extraction is to enforce the prover to send the encryption of witnesses (with a public key given in the CRS) along with proof, so that in security proofs the extractor can use the pair secret key for extraction [Gro06]. Using this technique, the proof (communication) size will not be succinct anymore, as impossibility result in [GW11] confirms, but the verification will be efficient yet and the extraction issue that zk-SNARKs have in the UC framework [Can01] will be solved.

3.2 Preliminaries

3.2.1 Notations

Let PPT denote probabilistic polynomial-time, and NUPPT denote non-uniform PPT. Let $\kappa \in \mathbb{N}$ be the security parameter, say $\kappa = 128$. All adversaries will be stateful. For an algorithm $\mathcal{A}$, let $\mathrm{im}(\mathcal{A})$ be the image of $\mathcal{A}$, i.e., the set of valid outputs of $\mathcal{A}$, let $\mathsf{RND}(\mathcal{A})$ denote the random tape of $\mathcal{A}$, and let $r \leftarrow \mathsf{RND}(\mathcal{A})$ denote sampling of a randomizer r of sufficient length for $\mathcal{A}$'s needs. By $y \leftarrow \mathcal{A}(x; r)$ we mean given an input x and a randomizer r, $\mathcal{A}$ outputs y. For algorithms $\mathcal{A}$ and $\mathsf{Ext}_{\mathcal{A}}$, we write $(y \parallel y') \leftarrow (\mathcal{A} \parallel \mathsf{Ext}_{\mathcal{A}})(x; r)$ as a shorthand for "$y \leftarrow \mathcal{A}(x; r)$, $y' \leftarrow \mathsf{Ext}_{\mathcal{A}}(x; r)$". An arbitrary negligible function is shown with negl. Two computationally indistinguishable distributions A and B are shown with $A \approx_c B$.

In pairing-based groups, we use additive notation together with the bracket notation, i.e., in group $\mathbb{G}_1$, $[a]_1 =$

$a\,[1]_1$, where $[1]_1$ is a fixed generator of $\mathbb{G}_1$. A *bilinear group generator* $\mathsf{BGgen}(1^\kappa)$ returns $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}, [1]_1, [1]_2)$, where p (a large prime) is the order of cyclic abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$. Finally, $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an efficient non-degenerate bilinear pairing, s.t. $\hat{e}([a]_1, [b]_2) = [ab]_T$. Denote $[a]_1 \bullet [b]_2 = \hat{e}([a]_1, [b]_2)$.

We bellow review Square Arithmetic Programs (SAPs) that defines NP-complete language specified by a quadratic equation over polynomials [GM17].

Square Arithmetic Program: Any quadratic arithmetic circuit with fan-in 2 gates over a finite field $\mathbb{Z}_p$ can be lifted to a SAP instance over the same finite field (e.g. by considering $ab = ((a+b)^2 - (a-b)^2)/4$) [GM17]. A SAP instance contains $\mathcal{S}_p = (\mathbb{Z}_p, m_0, \{u_j, w_j\}_{j=0}^m)$. This instance defines the following relation:

$$\mathbf{R}_{\mathcal{S}_p} = \left\{ \begin{array}{l} (\mathsf{x}, \mathsf{w})\colon \mathsf{x} = (A_1, \ldots, A_{m_0})^\top \wedge \mathsf{w} = (A_{m_0+1}, \ldots, A_m)^\top \wedge \\ \left(\sum_{j=0}^m A_j u_j(X)\right)^2 \equiv \sum_{j=0}^m A_j w_j(X) \pmod{\ell(X)} \end{array} \right\}$$

where $\ell(X) := \prod_{i=1}^n (X - \omega^{i-1}) = X^n - 1$ is the unique degree n monic polynomial such that $\ell(\omega^{i-1}) = 0$ for all $i \in [1 \,..\, n]$. Alternatively, $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}_{\mathcal{S}_p}$ if there exists a (degree $\leq n-2$) polynomial $h(X)$, s.t. $\left(\sum_{j=0}^m A_j u_j(X)\right)^2 - \sum_{j=0}^m A_j w_j(X) = h(X)\ell(X)$.

3.2.2 Definitions

We use the definitions of NIZK arguments from [Gro06,Gro16,GM17,KZM+15a]. Let $\mathcal{R}$ be a relation generator, such that $\mathcal{R}(1^\kappa)$ returns a polynomial-time decidable binary relation $\mathbf{R} = \{(\mathsf{x}, \mathsf{w})\}$. Here, x is the statement and w is the witness. We assume one can deduce κ from the description of $\mathbf{R}$. The relation generator also outputs auxiliary information $\xi_{\mathbf{R}}$ that will be given to the honest parties and the adversary. As in [Gro16, ABLZ17], $\xi_{\mathbf{R}}$ is the value returned by $\mathsf{BGgen}(1^\kappa)$. Due to this, we also give $\xi_{\mathbf{R}}$ as an input to the honest parties; if needed, one can include an additional auxiliary input to the adversary. Let $\mathbf{L}_{\mathbf{R}} = \{\mathsf{x} : \exists \mathsf{w}, (\mathsf{x}, \mathsf{w}) \in \mathbf{R}\}$ be an NP language.

As a particular case of subversion-resistant NIZK arguments defined in section 2.2, a *NIZK argument system* Ψ for $\mathcal{R}$ consists of tuple of PPT algorithms, s.t.:

- **CRS generator**: K is a PPT algorithm that given $(\mathbf{R}, \xi_{\mathbf{R}})$, where $(\mathbf{R}, \xi_{\mathbf{R}}) \in \mathrm{im}(\mathcal{R}(1^\kappa))$ outputs $\mathsf{crs} = (\mathsf{crs}_P, \mathsf{crs}_V)$ and stores trapdoors of crs as τ. We distinguish crs_P (needed by the prover) from crs_V (needed by the verifier).
- **Prover**: P is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_P, \mathsf{x}, \mathsf{w})$, where $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$, outputs an argument π. Otherwise, it outputs $\perp$.
- **Verifier**: V is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_V, \mathsf{x}, \pi)$, returns either 0 (reject) or 1 (accept).
- **Simulator**: Sim is a PPT algorithm that, given $(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \tau, \mathsf{x})$, outputs an argument π.
- **Extractor**: Ext is a PPT algorithm that, given $(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}, \mathsf{x}, \pi, \tau_e)$ extracts the w; where τ_e is extraction trapdoor (e.g. a secret key).

We require an argument system Ψ to be complete, computationally knowledge-sound and statistically ZK, as in the following definitions.

Perfect Completeness [Gro16]: A non-interactive argument Ψ is *perfectly complete for* $\mathcal{R}$, if for all κ, all $(\mathbf{R}, \xi_{\mathbf{R}}) \in RANGE(\mathcal{R}(1^\kappa))$, and $(\mathsf{x}, \mathsf{w}) \in \mathbf{R}$,

$$\Pr\left[\mathsf{crs} \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}) : \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \mathsf{P}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})) = 1\right] = 1 \ .$$

Computational Knowledge-Soundness [Gro16]: A non-interactive argument Ψ is computationally (adaptively) *knowledge-sound for* $\mathcal{R}$, if for every NUPPT $\mathcal{A}$, there exists a NUPPT extractor $\mathsf{Ext}_{\mathcal{A}}$, s.t. for all κ,

$$\Pr\left[\begin{array}{l}(\mathbf{R}, \xi_{\mathbf{R}}) \leftarrow \mathcal{R}(1^\kappa), (\mathsf{crs} \parallel \tau) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}), \\ r \leftarrow_r \mathsf{RND}\mathcal{A}, ((\mathsf{x}, \pi) \parallel \mathsf{w}) \leftarrow (\mathcal{A} \parallel \mathsf{Ext}_{\mathcal{A}})(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}; r) : \\ (\mathsf{x}, \mathsf{w}) \notin \mathbf{R} \wedge \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi) = 1\end{array}\right] \approx_\kappa 0 \enspace .$$

Here, $\xi_{\mathbf{R}}$ can be seen as a common auxiliary input to $\mathcal{A}$ and $\mathsf{Ext}_{\mathcal{A}}$ that is generated by using a benign [?] relation generator; A knowledge-sound argument system is called an *argument of knowledge*.
Statistically Zero-Knowledge [Gro16]: A non-interactive argument Ψ is *statistically ZK for* $\mathcal{R}$, if for all κ, all $(\mathbf{R}, \xi_{\mathbf{R}}) \in RANGE(\mathcal{R}(1^\kappa))$, and for all NUPPT $\mathcal{A}$, $\varepsilon_0^{unb} \approx_\kappa \varepsilon_1^{unb}$, where

$$\varepsilon_b = \Pr[(\mathsf{crs} \parallel \tau) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}) : \mathcal{A}^{\mathsf{O}_b(\cdot,\cdot)}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}) = 1] \enspace .$$

Here, the oracle $\mathsf{O}_0(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R}$, and otherwise it returns $\mathsf{P}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{P}}, \mathsf{x}, \mathsf{w})$. Similarly, $\mathsf{O}_1(\mathsf{x}, \mathsf{w})$ returns $\perp$ (reject) if $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R}$, and otherwise it returns $\mathsf{Sim}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \mathsf{x}, \tau)$. Ψ is *perfect ZK for* $\mathcal{R}$ if one requires that $\varepsilon_0 = \varepsilon_1$.

Intuitively, a non-interactive argument Ψ is zero-knowledge if it does not leak extra information besides the truth of the statement. Beside the mentioned properties defined in Def. 3.2.2-3.2.2, a zk-SNARK has *succinctness* property, meaning that the proof size is $\mathsf{poly}(\kappa)$ and the verifier's computation is $\mathsf{poly}(\kappa)$ and the size of instance. In the rest, we recall the definitions of simulation soundness and simulation extractability that are used in construction of UC-secure zk-SNARKs.
Simulation Soundness [Gro06]: A non-interactive argument Ψ is *simulation sound for* $\mathcal{R}$ if for all NUPPT $\mathcal{A}$, and all κ,

$$\Pr\left[\begin{array}{l}(\mathbf{R}, \xi_{\mathbf{R}}) \leftarrow \mathcal{R}(1^\kappa), (\mathsf{crs} \parallel \tau) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}), (\mathsf{x}, \pi) \leftarrow \mathcal{A}^{o(\cdot)}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}) : \\ (\mathsf{x}, \pi) \notin Q \wedge \mathsf{x} \notin \mathbf{L} \wedge \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi) = 1\end{array}\right] \approx_\kappa 0 \enspace .$$

Here, Q is the set of simulated statement-proof pairs generated by adversary's queries to o, that returns simulated proofs.
Non-Black-Box Simulation Extractability [GM17]: A non-interactive argument Ψ is *non-black-box simulation-extractable for* $\mathcal{R}$, if for any NUPPT $\mathcal{A}$, there exists a NUPPT extractor $\mathsf{Ext}_{\mathcal{A}}$ s.t. for all κ,

$$\Pr\left[\begin{array}{l}(\mathbf{R}, \xi_{\mathbf{R}}) \leftarrow \mathcal{R}(1^\kappa), (\mathsf{crs} \parallel \tau) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}), \\ r \leftarrow_r \mathsf{RND}\mathcal{A}, ((\mathsf{x}, \pi) \parallel \mathsf{w}) \leftarrow (\mathcal{A}^{o(\cdot)} \parallel \mathsf{Ext}_{\mathcal{A}})(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}; r) : \\ (\mathsf{x}, \pi) \notin Q \wedge (\mathsf{x}, \mathsf{w}) \notin \mathbf{R} \wedge \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi) = 1\end{array}\right] \approx_\kappa 0 \enspace .$$

Here, Q is the set of simulated statement-proof pairs generated by adversary's queries to o that returns simulated proofs. It is worth to mention that *non-black-box simulation extractability* implies *knowledge soundness* (given in Def. 3.2.2), as the earlier is a strong notion of the later which additionally the adversary is allowed to send query to the proof simulation oracle. Similarly, one can observe that *non-black-box simulation extractability* implies *simulation soundness* (given in Def. 3.2.2) that is discussed in [Gro06] with more details.
Black-Box Simulation Extractability [KZM+15a]: A non-interactive argument Ψ is *black-box simulation-extractable for* $\mathcal{R}$ if there exists a black-box extractor Ext that for all NUPPT $\mathcal{A}$, and all κ,

$$\Pr\left[\begin{array}{l}(\mathbf{R}, \xi_{\mathbf{R}}) \leftarrow \mathcal{R}(1^\kappa), (\mathsf{crs} \parallel \tau_s \parallel \tau_e) \leftarrow \mathsf{K}(\mathbf{R}, \xi_{\mathbf{R}}), \\ (\mathsf{x}, \pi) \leftarrow \mathcal{A}^{o(\cdot)}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}), \mathsf{w} \leftarrow \mathsf{Ext}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}, \tau_e, \mathsf{x}, \pi) : \\ (\mathsf{x}, \pi) \notin Q \wedge (\mathsf{x}, \mathsf{w}) \notin \mathbf{R} \wedge \mathsf{V}(\mathbf{R}, \xi_{\mathbf{R}}, \mathsf{crs}_{\mathsf{V}}, \mathsf{x}, \pi) = 1\end{array}\right] \approx_\kappa 0 \enspace .$$

Similarly, Q is the set of simulated statement-proof pairs, and τ_e is the extraction trapdoor. A key note about Def. 3.2.2 is that the extraction procedure is black-box and unlike the non-black-box case, the extractor Ext works for all adversaries.

3.2.3 COCO: a Framework for Constructing UC-secure zk-SNARKs

Kosba et al. [KZM+15a] have constructed a framework with several converters which the most powerful one gets a sound NIZK and lifts to a NIZK that achieves *black-box simulation extractability* (defined in Def. 3.2.2), or equivalently UC-security [Gro06]. Here we review construction of the most powerful converter that is used by both Hawk and Gyges to construct a UC-secure zk-SNARK. Note that in this case the proofs are succinct in the circuit size and linear in the witness size.

Construction. Given a sound NIZK, to achieve a UC-secure NIZK, COCO framework applies several changes in all setup, proof generation and verification procedures of the input NIZK. Initially the framework defines a new language $\mathbf{L}'$ based on the language $\mathbf{L}$ in underlying NIZK and some new primitives that are needed for the transformation. Let $(\mathsf{KGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ be a set of algorithms for a semantically secure encryption scheme, $(\mathsf{KGen}_s, \mathsf{Sig}_s, \mathsf{vfy}_s)$ be a one-time signature scheme and $(\mathsf{Com}_c, \mathsf{vfy}_c)$ be a perfectly binding commitment scheme.

Given a language $\mathbf{L}$ with the corresponding NP relation $\mathbf{R_L}$, define a new language $\mathbf{L}'$ such that $((\mathsf{x}, c, \mu, pk_s, pk_e, \rho), (r, r_0, \mathsf{w}, s_0)) \in \mathbf{R_{L'}}$ iff:

$$(c = \mathsf{Enc}_e(pk_e, \mathsf{w}; r)) \wedge ((\mathsf{x}, \mathsf{w}) \in \mathbf{R_L} \vee (\mu = f_{s_0}(pk_s) \wedge \rho = \mathsf{Com}_c(s_0; r_0)))\ ,$$

where $\{f_s : \{0,1\}^* \to \{0,1\}^\kappa\}_{s \in \{0,1\}^\kappa}$ is a pseudo-random function family. Now, a sound NIZK argument system Ψ for $\mathcal{R}$ constructed from PPT algorithms $(\mathsf{K}, \mathsf{P}, \mathsf{V}, \mathsf{Sim}, \mathsf{Ext})$ can be lifted to a UC-secure NIZK Ψ' with PPT algorithms $(\mathsf{K}', \mathsf{P}', \mathsf{V}', \mathsf{Sim}', \mathsf{Ext}')$ as follows.

CRS and trapdoor generation $\mathsf{K}'(\mathbf{R_L}, \xi_{\mathbf{R_L}})$**:** Sample $(\mathsf{crs} \parallel \tau) \leftarrow \mathsf{K}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}})$; $(pk_e, sk_e) \leftarrow \mathsf{KGen}_e(1^\kappa)$; $s_0, r_0 \leftarrow \{0,1\}^\kappa$; $\rho := \mathsf{Com}_c(s_0; r_0)$; and output $(\mathsf{crs}' \parallel \tau' \parallel \tau'_e) := ((\mathsf{crs}, pk_e, \rho) \parallel (s_0, r_0) \parallel sk_e)$.

Prover $\mathsf{P}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}, \mathsf{x}, \mathsf{w})$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e, \rho)$; Abort if $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R_L}$; $(pk_s, sk_s) \leftarrow \mathsf{KGen}_s(1^\kappa)$; sample $z_0, z_1, z_2, r_1 \leftarrow \{0,1\}^\kappa$; compute $c = \mathsf{Enc}_e(pk_e, \mathsf{w}; r_1)$; generate $\pi \leftarrow \mathsf{P}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, z_0, pk_s, pk_e, \rho), (r_1, z_1, w, z_2))$; sign $\sigma \leftarrow \mathsf{Sig}_s(sk_s, (\mathsf{x}, c, z_0, \pi))$; and output $\pi' := (c, z_0, \pi, pk_s, \sigma)$.

Verifier $\mathsf{V}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \mathsf{x}, \pi')$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e, \rho)$ and $\pi' := (c, \mu, \pi, pk_s, \sigma)$; Abort if $\mathsf{vfy}_s(pk_s, (\mathsf{x}, c, \mu, \pi), \sigma) = 0$; call $\mathsf{V}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, \mu, pk_s, pk_e, \rho), \pi)$ and abort if it outputs 0.

Simulator $\mathsf{Sim}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \tau', \mathsf{x})$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e, \rho)$ and $\tau' := (s_0, r_0)$; $(pk_s, sk_s) \leftarrow \mathsf{KGen}_s(1^\kappa)$; set $\mu = f_{s_0}(pk_s)$; sample $z_3, r_1 \leftarrow \{0,1\}^\kappa$; compute $c = \mathsf{Enc}_e(pk_e, z_3; r_1)$; generate $\pi \leftarrow \mathsf{P}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, \mu, pk_s, pk_e, \rho), (r_1, r_0, z_3, s_0))$; sign $\sigma \leftarrow \mathsf{Sig}_s(sk_s, (\mathsf{x}, c, \mu, \pi))$; and output $\pi' := (c, \mu, \pi, pk_s, \sigma)$.

Extractor $\mathsf{Ext}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \tau', \mathsf{x}, \pi')$**:** Parse $\pi' := (c, \mu, \pi, pk_s, \sigma)$, $\tau' := sk_e$; extract $\mathsf{w} \leftarrow \mathsf{Dec}_e(sk_e, c)$; output w.

3.3 Efficient UC-secure zk-SNARKs

In this section, we show that given a non-black-box simulation-sound NIZK, one can construct black-box simulation-extractable NIZK by minimal changes. To do so, as an instance, we present a variation of GM zk-SNARK [GM17] and show that it achieves black-box simulation extractability, and equivalently UC-security. Note that the requirement that the input NIZK archives non-black-box simulation-soundness is the main reason that allows us to achieve black-box simulation extractability with minimal changes. GM zk-SNARK is the first scheme that guarantees non-black-box simulation extractablity and this is one reason that we use it in our instantiation.

Intuition. The goal is to present a UC-secure zk-SNARK but more efficient than UC-secure zk-SNARKs that are lifted by COCO framework; especially more efficient than the ones that are deployed in [KMS+15, JKS16]. To do so, we slightly modify the input zk-SNARK that guarantees non-black-box simulation extractability (e.g. GM zk-SNARK) and enforce prover P to encrypt its witnesses with a public key given in the CRS and send the ciphertext along with the proof. In this scenario, in security proof, the secret key of encryption scheme is given to the Ext which allows to extract witnesses in black-box manner, that is more realistic indeed. Actually this is an already known technique to achieve black-box extraction that also is used in COCO framework. It is undeniable that sending encryption of witnesses leads to have non-succinct proofs in witness size but still they are succinct in the size of circuit that encodes the language and it is simpler and for particular settings more efficient than the ones that are lifted by COCO.

3.3.1 Construction

While modifying we keep internal computation of both prover and verifier as original one, that considerably are optimized for a SAP relation. Instead we define a new language $\mathbf{L}'$ based on the language $\mathbf{L}$ in the input NIZK (here GM zk-SNARK) that is embedded with encryption of witness. Strictly speaking, given a language $\mathbf{L}$ with the corresponding NP relation $\mathbf{R_L}$, we define the following new language $\mathbf{L}'$ such that $((\mathsf{x}, c, pk_e), (\mathsf{w}, r)) \in \mathbf{R_{L'}}$ iff:

$$(c = \mathsf{Enc}_e(pk_e, \mathsf{w}; r)) \wedge ((\mathsf{x}, \mathsf{w}) \in \mathbf{R_L}),$$

where $(\mathsf{KGen}_e, \mathsf{Enc}_e, \mathsf{Dec}_e)$ is a set of algorithms for a semantically secure encryption scheme with keys (pk_e, sk_e). Accordingly, the modified version of GM zk-SNARK is given in Fig. 3.1. It is worth to mention that, due to the particular structure of new language $\mathbf{L}'$, all verifications will be done inside the circuit, and interestingly verifier and prover's internal computations are the same as before, just prover needs to send encryption of witnesses along with the proof. This is the key modification in removing nBB extraction (particularly knowledge-assumption based in zk-SNARKs) and achieving BB extraction.

CRS and trapdoor generation $\mathsf{K}'(\mathbf{R_L}, \xi_{\mathbf{R_L}})$**:** Generate key pair $(pk_e, sk_e) \leftarrow \mathsf{KGen}_e(1^\kappa)$; execute CRS generator of GM zk-SNARK and sample $(\mathsf{crs} \,\|\, \tau) \leftarrow \mathsf{K}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}})$; output $(\mathsf{crs}' \,\|\, \tau' \,\|\, \tau'_e) := ((\mathsf{crs}, pk_e) \,\|\, \tau \,\|\, sk_e)$; where τ' are simulation trapdoors and τ'_e is the extraction trapdoor (key).

Prover $\mathsf{P}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \mathsf{x}, \mathsf{w})$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e)$; Abort if $(\mathsf{x}, \mathsf{w}) \notin \mathbf{R_L}$; sample $r \leftarrow \{0,1\}^\kappa$; compute encryption of witnesses $c = \mathsf{Enc}_e(pk_e, \mathsf{w}; r)$; execute prover P of GM zk-SNARK and generate $\pi \leftarrow \mathsf{P}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, pk_e), (\mathsf{w}, r))$; and output $\pi' := (c, \pi)$.

Verifier $\mathsf{V}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \mathsf{x}, \pi')$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e)$ and $\pi' := (c, \pi)$; call verifier $\mathsf{V}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, pk_e), \pi)$ of GM zk-SNARK and abort if it rejects.

Simulator $\mathsf{Sim}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \mathsf{x}, \tau')$**:** Parse $\mathsf{crs}' := (\mathsf{crs}, pk_e)$ and $\tau' := \tau$; sample $z, r \leftarrow \{0,1\}^\kappa$; compute $c = \mathsf{Enc}_e(pk_e, z; r)$; execute simulator of GM zk-SNARK and generate $\pi \leftarrow \mathsf{Sim}(\mathbf{R_{L'}}, \xi_{\mathbf{R_{L'}}}, \mathsf{crs}, (\mathsf{x}, c, pk_e), \tau)$; and output $\pi' := (c, \pi)$.

Extractor $\mathsf{Ext}'(\mathbf{R_L}, \xi_{\mathbf{R_L}}, \mathsf{crs}', \tau', \mathsf{x}, \pi')$**:** Parse $\pi' := (c, \pi)$ and $\tau' := sk_e$; extract $\mathsf{w} \leftarrow \mathsf{Dec}_e(sk_e, c)$; output w.

Figure 3.1: Transformation of GM zk-SNARK to achieve black-box simulation extractability

3.3.2 Efficiency

With the proposed transformation the computations of prover and verifier will ba as original one, but for an instance with larger size. The communication size will be extended to linear size but interestingly the proof still will be succinct. Note that the linear communication size will be added to the statement. In the rest, we particularly talk abut the performance of transformation while instantiated with Groth and Maller's zk-SNARK.

In the lifted version of GM zk-SNARK, as the original one, proof is 2 elements from $\mathbb{G}_1$ and 1 element from $\mathbb{G}_2$, but along with c that is encryption of witnesses. So, communication is dominated with size of c that is linear in witness size (this can be considered as commitment in the commit-and-proof systems) but proof elements are only 3 group elements.

As verifier is untouched, so similar to GM zk-SNARK, the verification procedure consists of checking that the proof contains 3 appropriate group elements and checking 2 pairing product equations which in total it needs a multi-exponentiation $\mathbb{G}_1$ to m_0 exponents and 5 pairings.

In the setup, in result of our change, the arithmetic circuit will be slightly extended, but due to minimal changes (a more detailed comparison is provided in Fig. 3.2), the extension is less than the case that one uses COCO framework.

3.3.3 Security Proof

Theorem 3 (Perfect Completeness)**.** *The protocol constructed in Sec. 3.3, is a non-interactive argument of knowledge that guarantees perfect completeness.*

Proof. Proof of theorem is described in [Bag19a]. □

Theorem 4 (Computationally Zero-Knowledge)**.** *The protocol constructed in Sec. 3.3, is a non-interactive argument of knowledge that guarantees computational zero-knowledge.*

Proof. Proof of theorem is described in [Bag19a]. □

Theorem 5 (Black-Box Simulation Extractability)**.** *Assuming the encryption scheme is semantically secure and perfectly correct, the modified version of GM zk-SNARK in Sec. 3.3, satisfies black-box simulation extractability.*

Proof. Proof of theorem is described in [Bag19a]. □

3.4 On the Efficiency of Privacy Preserving Smart Contract Systems

Both the privacy-preserving smart contract systems Hawk and Gyges [KMS+15, JKS16] frequently generate CRS and use a UC-secure zk-SNARK to prove different statements. In Hawk author discuss that their system is dominated by efficiency of the underlying UC-secure zk-SNARK that are achieved from a variation of Pinocchio zk-SNARK [PHGR13] lifted by COCO framework (the same is done in Gyges as well). In the rest, we discuss how the simplified transformation described in Sec. 3.3 can help to improve the efficiency of both smart contract systems. Our evaluation is focused precisely on Hawk, but as Gyges also have used COCO framework, so the same evaluation can be considered for Gyges.

On the Efficiency of Hawk. We begin evaluation of Hawk by reviewing the changes that are applied on Ben Sasson et al.'s zk-SNARK (to get UC-security) before using it in Hawk. As discussed in Sec. 3.2.3, in order to lift any sound NIZK to a UC-secure NIZK, COCO applies several changes in setup, proof generation and proof verification of input NIZK. For instance, each time prover needs to generate a pair of signing/verifying keys for a one-time secure signature scheme, encrypt the witnesses using a given public-key, and sign the generated proof using the mentioned one-time signing key. On the other side, verifier needs to do extra verifications than the NIZK verification.

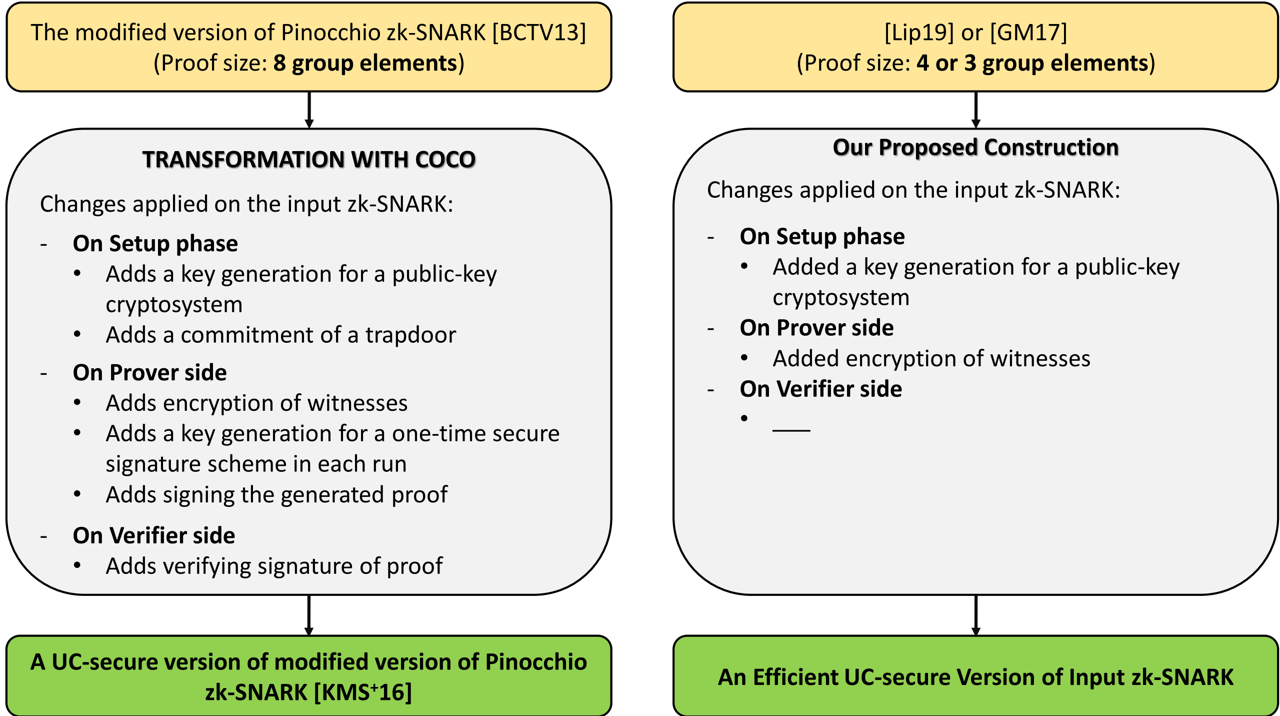


Figure 3.2: The modifications applied by COCO transformation on the modified version of Pinocchio zk-SNARK [BCTV13] before using in Hawk system versus our changes on a simulation sound NIZK (particularly on GM zk-SNARK, shown in Fig. 3.1) to get a UC-secure NIZK.

Table 3.1: Comparison of Ben Sasson et al.'s [BCTV13] and GM [GM17] zk-SNARKs for arithmetic circuit satisfiability with m_0 element instance, m wires, n multiplication gates. Since [GM17] uses squaring gates, so n multiplication gates translate to $2n$ squaring gates. Implementations are done on a PC with 3.40 GHz Intel Core i7-4770 CPU, in single-threaded mode, for an R1CS instance with $n = 10^6$ constraints and $m = 10^6$ variables, of which $m_0 = 10$ are input variables. $\mathbb{G}_1$ and $\mathbb{G}_2$: group elements, E: exponentiations and P: pairings.

zk-SNARKs	CRS Length, Generation Time	Proof Size	Computation of P	Computation of V	Verification Equ.
[BCTV13] &	$6m + n - m_0$ $\mathbb{G}_1$ m $\mathbb{G}_2$	7 $\mathbb{G}_1$ 1 $\mathbb{G}_2$	$6m + n - m_0$ E_1 m E_2	m_0 E_1 12 P	5
in libsnark	104.8 seconds	287 bytes	128.6 seconds	4.2 millisec.	—
[GM17] &	$m + 4n + 5$ $\mathbb{G}_1$ $2n + 3$ $\mathbb{G}_2$	2 $\mathbb{G}_1$ 1 $\mathbb{G}_2$	$m + 4n - m_0$ E_1 $2n$ E_2	m_0 E_1 5 P	2
in libsnark	100.4 seconds	127 bytes	116.4 seconds	2.3 millisec.	—

As we discussed in Sec. 3.3, to achieve a UC-secure version of a simulation sound NIZK (in our case GM zk-SNARK), we added a key generation procedure for a public-key cryptosystem in the setup phase, and prover only needed to encrypt the witnesses using the public-key in CRS and then generate a proof for new language as the original NIZK. We did not add new checking to the verifier side and it is as the non-UC-secure version.

Left side of Fig. 3.2 summarizes the modifications applied (by using COCO) on a variation of Pinocchio zk-SNARK before using in Hawk; and right side summarizes our required changes on a simulation sound NIZK (e.g. on [Lip19] or [GM17]) to get BB simulation extractability and equivalently UC-security. As both use encrypting of witnesses, it seems having linear proof size on witness size currently is an undeniable issue to get black-box extraction. So, except this unavoidable modification, we applied minimal changes in the structure of

GM zk-SNARK to achieve a UC-secure version of it.

Additionally, Tab.3.1 compares efficiency and practical performance of Ben Sasson et al.'s [BCTV13] and GM [GM17] zk-SNARKs from various perspectives before applying any changes. Empirical performance reported in libsnark library for a particular instance[2]. Following Pinocchio scheme, Ben Sasson et al.'s zk-SNARK [BCTV13] is constructed for the QAP relation, while Groth and Maller's scheme works for the SAP relation by default. As discussed in [Gro16, GM17], a SAP instance can be constructed based on a simplification of systems on arithmetic constraints, such that all multiplication gates are replaced with squaring gates, but with at most two times gates.

Tab. 3.1 shows that GM zk-SNARK outperforms Ben Sasson et al.'s zk-SNARK in all metrics for a circuit with 10^6 gates. Beside faster running times in all algorithms, GM zk-SNARK has only 2 verification equations, instead of 5 in [BCTV13]. By considering efficiency report in Tab.3.1, and the fact that our modifications (summarized in Fig. 3.2) are lighter than what are applied on Ben Sasson et al.'s zk-SNARK before deploying in Hawk system, one can observe that for circuits with smaller number of gates new UC-secure zk-SNARK will simplify the system and would be more efficient than the one that currently is used in Hawk (similarly in Gyges). Indeed our changes are a small part of their already applied changes, so they will have less overhead.

An important note is that GM zk-SNARK is constructed for SAP relation while the currently deployed zk-SNARK in both smart contract systems is constructed for QAP relation. Due to this fact, with our installation we expect to have better efficiency in circuits with smaller number of gates. In circuits with larger number of gates, to achieve better efficiency one can use one of QAP-based schemes proposed in [Lip19, AB19, Bag19b] that achieve zero-knowledge even their public parameters are computed maliciously.

Hawk needs to generate CRS of zk-SNARK for each smart contract and as the UC-secure zk-SNARK is widely deployed in various operations of the system, so by substituting a zk-SNARK constructed using the transformation discussed in Sec. 3.3, one can simplify the system and improve the efficiency of whole system.

Moreover, in the construction of the Hawk system, authors applied various effective optimizations to maximize the efficiency of underlying UC-secure zk-SNARK (Sec. V in [KMS+15]). The same techniques can work with our new construction. For instance, it is shown that in the *Finalize* operation of a smart contract in Hawk, one may use non-UC-secure zk-SNARK, which similarly in new case one can use non-UC-secure version of GM zk-SNARK that is more efficient than the one that currently is used (compared in Tab. 3.1) and additionally it ensures non-block-box simulation extractability. In another noticeable optimization, Kosba et al. used some independently optimized primitives in the lifted UC-secure zk-SNARK, that had considerable effect in the practical efficiency of Hawk. Again, by reminding that our changes are a small part of the changes applied by COCO, so a part of their optimized primitives (for encryption scheme) can be used in this case as well, but the rest can be ignored.

[2] Based on reported implementation on https://github.com/scipr-lab/libsnark

Chapter 4

Bulletin Board for E-voting

4.1 Introduction

Most of the E-voting literature assumes that the election data is stored either in a single server or that there exists a distributed ledger, typically called a bulletin board (BB). BB is typically treated as a black box and very few works have studied special requirements and constructions of e-voting BBs in detail. For example, compared to standard ledgers, in e-voting scenario voter might need to get back a receipt guaranteeing that their vote was safely stored. This is especially important in e-voting systems where the bulletin board is not public to everyone and voter has no way to directly check that the vote was stored.

4.2 Preliminaries

One of the most promising of the existing candidates is the bulletin board construction by Culnane and Schneider [CS14b]. We studied their protocol and in doing so notice some vulnerabilities. Most importantly, we showed that given that less than $N/3$ of the N bulletin board peers are corrupted, the protocol might not terminate. This contrary to the claims in their paper.

We proposed the first cryptographic security definition for e-voting bulletin boards which is captured in two properties: (i) *confirmable liveness* – meaning that BB protocol will eventually terminate and an any honest voter will be provided with a valid receipts, and (ii) *(confirmable) persistence* – it is impossible to remove items from BB and items can be posted only through legitimate posting procedure. Then, taking Culnane and Schneider's protocol as a basis, we proposed a new BB protocol satisfying this security definition. We proved that it tolerates any number less than $N/3$ out-of N corrupted bulletin board peers both for persistence and confirmable liveness, against a computationally bounded general Byzantine adversary. Furthermore, *persistence* can also be made confirmable, in the sense that any malicious behavior can be detected via a verification mechanism, if we distribute the audit board (this is where the output of the BB protocol is eventually stored) as a replicated service with honest majority.

4.3 Our contribution

Our contribution was made in [KKL+18] and it has two main parts.

First, we point out an attack against the scheme presented in [CS14b]. Second, we propose a scheme that fixes this problem. The main idea is that malicious peers are forced to either reveal themselves in which case they can be ignored, or to behave benignly.

4.3.1 Attacking the liveness of the CS BB system

As informally argued in [CS14a, Sec. 8] (the full version of [CS14b]), the liveness in CS can be achieved if one of the following conditions hold: 1) all the peers are following the protocol honestly and are online, 2) a threshold of $t_c < N_c/3$ peers is malicious, but all users are honest, or 3) the more general condition that not all users are honest and *the malicious peers may choose any database in their capability, but do not change their database once it has been fixed, and will not send different databases to different peers*. The argument is that one can easily detect in practice if malicious peers send different databases to different peers.

We demonstrate an attack against the Confirmable Liveness of CS in our framework. Although our attack falls outside the threat model of [CS14b], it reveals that the presumed "fear of detection" that justifies the said threat model, and especially the more general condition 3) described above, is not rigorously addressed. In particular, we show that the liveness adversary may choose to split the honest peers into two groups, and yet not be detected by being consistent w.r.t. to the peers in the same group. This way, the adversary manages a liveness breach, while the honest IC peers cannot detect the attack relying on the protocol guidelines and their local views. As a result, our attack shows that the original description of CS must be enhanced with an explicit detection mechanism against any deviation from the IC consensus protocol specifications, in order for the threat model in [CS14b] to be properly justified. On the other hand, as we describe in Section 4.3.2 and prove in [KKL+18], enhancing CS with our novel **Publishing** protocol completely overcomes such issues, by achieving Confirmable Liveness even against a general Byzantine adversary.

Description of the liveness attack. Our attack works under fault tolerance threshold $N_c > 3t_c$ (where t_c denotes the threshold of malicious peers), as required in [CS14b], and consists of the steps below.

STEP 1: Let p be a period where the set of honestly posted items is non-empty. For simplicity, we assume that there is a single honest user U_h who broadcasts x_h to all IC peers P_i, $i \in [N_c]$, and obtains a valid receipt $\mathsf{rec}[x_h]$.

STEP 2: A malicious user U_c deviates from broadcasting and sends x_c to all t_c corrupted peers and $N_c - 2t_c$ honest peers. Denote the latter set of honest $N_c - 2t_c$ peers by $\mathcal{H}_{\mathsf{in}}$. The malicious peers engage in the **Posting** protocol by interacting only with the peers in $\mathcal{H}_{\mathsf{in}}$. Observe that even if t_c honest peers do not participate in the post request of x_c, the collaboration of $t_c + (N_c - 2t_c) = N_c - t_c$ peers is enough so that U_c obtains a valid receipt $\mathsf{rec}[x_c]$, yet $(p, x_c) \in B_{i,p}$ only for honest peers $P_i \in \mathcal{H}_{\mathsf{in}}$. Denote by $\mathcal{H}_{\mathsf{out}}$ the t_c honest peers s.t. $x_c \notin B_{i,p}$.

STEP 3: Another malicious user $\hat{U}_c$ deviates from broadcasting and, like U_c, sends item $\hat{x}_c$ to all t_c corrupted peers and the $N_c - 2t_c$ honest peers in $\mathcal{H}_{\mathsf{in}}$. However, now the malicious peers do not engage in the **Posting** protocol, so the peers in $\mathcal{H}_{\mathsf{in}}$ do not suffice for a receipt for $\hat{x}_c$.

STEP 4: When **Publishing** protocol starts, the honest peers in $\mathcal{H}_{\mathsf{in}}$ and $\mathcal{H}_{\mathsf{out}}$ engage in the Optimistic protocol by sending their signed local records $\mathcal{R}_h^c := \{(p, x_h), (p, x_c)\}$ and $\mathcal{R}_h := \{(p, x_h)\}$ respectively. From their side, the malicious peers sign their records as $\mathcal{R}_h^{c,\hat{c}} := \{(p, x_h), (p, x_c), (p, \hat{x}_c)\}$. As a result, none of the three records $\mathcal{R}_h$, $\mathcal{R}_h^c$ and $\mathcal{R}_h^{c,\hat{c}}$ is signed by at least $N_c - t_c$ peers (recall that $|\mathcal{H}_{\mathsf{in}}| = N_c - 2t_c$ and $|\mathcal{H}_{\mathsf{out}}| = t_c$). Therefore, the malicious peers force all honest peers to engage in the Fallback protocol.

STEP 5: During Fallback, all honest peers exchange their collection of signatures. At this step, each peer in $\mathcal{H}_{\mathsf{in}}$ sends to each peer in $\mathcal{H}_{\mathsf{out}}$ (i) its signature on $(p, x_c), (p, x_h)$ and $(p, \hat{x}_c)$ and (ii) the t_c signatures on (p, x_c) that it received from the malicious peers. This way, each peer in $\mathcal{H}_{\mathsf{out}}$ receives $(N_c - 2t_c) + t_c = N_c - t_c$ signatures on (p, x_c) but only $N_c - 2t_c$ signatures on $(p, \hat{x}_c)$, so it updates its local record to $\mathcal{R}_h^c$. Malicious peers send their signatures on $(p, x_c), (p, x_h)$ and $(p, \hat{x}_c)$ only to the peers in $\mathcal{H}_{\mathsf{in}}$. Therefore, each peer collects $(N_c - 2t_c) + t_c = N_c - t_c$ signatures on $(p, \hat{x}_c)$ and updates its local record to $\mathcal{R}_h^{c,\hat{c}}$.

STEP 6: When the Fallback round above is completed, all peers restart the Optimistic protocol. However, now the peers in $\mathcal{H}_{\mathsf{in}}$ and $\mathcal{H}_{\mathsf{out}}$ send their signed local records $\mathcal{R}_h^{c,\hat{c}}$ and $\mathcal{R}_h^c$ respectively. The malicious peers resend their records $\mathcal{R}_h^{c,\hat{c}}$ only to the peers in $\mathcal{H}_{\mathsf{in}}$, which now have $N_c - t_c$ signatures on $\mathcal{R}_h^{c,\hat{c}}$. Thus, they finalize their engagement in the **Publishing** protocol for period p by sending their TSS shares for $\mathcal{R}_h^{c,\hat{c}}$ to the AB.

STEP 7: After forcing the peers in $\mathcal{H}_{\mathsf{in}}$ to termination, the malicious peers become inert. This causes the peers in $\mathcal{H}_{\mathsf{out}}$ to remain pending for a new Fallback round that no other peer will follow. Moreover, the AB can

not obtain $N_c - t_c$ TSS shares on some agreed record, and thus it can not publish anything. This violates the property (bb.2) in [CS14b], which dictates that since x_h is an honestly posted item that has a receipt, it must be published to the AB. Thus, liveness is breached.

4.3.2 A New Publishing Protocol for the CS BB System

We present a new **Publishing** protocol that, when combined with the CS **Posting** protocol, results in a BB system that achieves Confirmable Liveness in a partially synchronous and Persistence in an asynchronous model, against a general Byzantine adversary, assuming a threshold of $t_c < N_c/3$ corrupted IC peers. Persistence can also be Confirmable, if we distribute the AB subsystem such that no more than $t_w < N_w/2$ out of the N_w AB peers are corrupted, as in [?]. Namely, the distributed AB runs as a replication service; data posting is done by broadcasting to all AB peers, while data reading is done by honest majority.

The public parameters params include the identities of the IC and AB peers, the description of DS, TSS (described in [KKL+18]), a *collision resistant hash function (CRHF)* $H_\kappa(\cdot)$, and all public and verification keys. All peers know consecutive periods $p = [T_{\mathsf{begin},p}, T_{\mathsf{end},p}]$, as well as the following moments per period p: (a) a moment $T_{\mathsf{barrier},p} \in (T_{\mathsf{begin},p}, T_{\mathsf{end},p})$, when item collection stops and the **Publishing** protocol is initiated; (b) a moment $T_{\mathsf{publish},p} \in (T_{\mathsf{barrier},p}, T_{\mathsf{end},p})$, where the AB peers publish their records for period p, and (c) a moment $T_{\mathsf{request},p} \in (T_{\mathsf{barrier},p}, T_{\mathsf{publish},p})$, where IC peers force exchange of information to finalize their records. For each period p, the phases of the **Publishing** protocol are as follows:

■ **Initialization** phase: each IC peer P_i initializes the following vectors:

(i). Its *direct view* of local records, denoted by $\mathsf{View}_{i,p} := \langle \tilde{B}_{i,1,p}, \ldots, \tilde{B}_{i,N_c,p} \rangle$: namely, it sets $\tilde{B}_{i,j,p} \leftarrow \perp$, for $j \neq i$, and $\tilde{B}_{i,i,p} \leftarrow B_{i,p}$.

(ii). For every $j \in [N_c] \setminus \{i\}$, its *indirect view* of local records as provided by P_j, denoted by $\mathsf{View}_{i,j,p} := \langle \tilde{B}^i_{j,1,p}, \ldots, \tilde{B}^i_{j,N_c,p} \rangle$, by setting $\mathsf{View}_{i,j,p} \leftarrow \langle \perp, \ldots, \perp \rangle$.

(iii). A variable vector $\langle b_{i,1}, \ldots, b_{i,N_c} \rangle$, where $b_{i,j}$ is a value in $\{?, 0, 1\}$ that expresses *the opinion of P_i on the validity of P_j's behavior*. Initially, $b_{i,i}$ is fixed to 1, while for $j \neq i$, $b_{i,j}$ is set to the "pending" value '?'. When P_i fixes $b_{i,j}$ to 1/0 for all $j \in [N_c]$, it engages in the **Finalization** phase described shortly.

(iv). A vector $\langle d_{i,1}, \ldots, d_{i,N_c} \rangle$, where $d_{i,j}$ is *the number of P_i's (direct or indirect) views that agree on P_j's record*. Initially, $d_{i,j} = 0$, for $j \neq i$, and $d_{i,i} = 1$.

■ **Collection** phase: upon initialization, P_i signs its local record $B_{i,p}$, followed by a tag RECORD, and broadcasts $\big((\textsc{record}, B_{i,p}), \sigma_{\mathsf{sk}_i}(\textsc{record}, B_{i,p})\big)$ to all IC peers. Then, P_i updates its direct and indirect views of other IC peers' records and fixes its opinion bit for their behavior, depending on the number of consistent signed messages it receives on each peer's record. In particular,

– *When P_i receives a message $\big((\textsc{record}, R_{i,j,p}), \sigma_{\mathsf{sk}_j}(\textsc{record}, R_{i,j,p})\big)$ signed by peer P_j that was never received before*, then it acts as follows: if $R_{i,j,p}$ is formatted as a non-$\perp$ record and the "opinion" bit $b_{i,j}$ is not fixed (i.e. $b_{i,j} =$ '?'), then it checks if $\mathsf{vfy}_{\mathsf{pk}_j}\big((\textsc{record}, R_{i,j,p}), \sigma_{\mathsf{sk}_j}(\textsc{record}, R_{i,j,p})\big) = 1$. If the latter holds, then P_i operates according to either of the following two cases:

1. If $\tilde{B}_{i,j,p} \neq \perp$, then it marks P_j as malicious, that is, it sets $\tilde{B}_{i,j,p} \leftarrow \perp$ and fixes $b_{i,j}$ to 0. Observe that since P_j is authenticated (except from some $\mathsf{negl}(\kappa)$ error), it is safe for P_i to mark P_j as malicious, as an honest peer would never send two different versions of its local records.

2. If $\tilde{B}_{i,j,p} = \perp$, then P_i updates $\mathsf{View}_{i,p}$ as $\tilde{B}_{i,j,p} \leftarrow R_{i,j,p}$, and $\mathsf{View}_{i,j,p}$ as $\tilde{B}^i_{j,j,p} \leftarrow R_{i,j,p}$ and increases the $d_{i,j}$ by 1. Next, it signs and re-broadcasts to all IC peers the received message in the format $\big(V_{i,j}, \sigma_{\mathsf{sk}_i}(V_{i,j})\big)$, where $V_{i,j} := \big((\textsc{view}, j), ((\textsc{record}, \tilde{B}_{i,j,p}), \sigma_{\mathsf{sk}_j}(\textsc{record}, \tilde{B}_{i,j,p}))\big)$. Upon fixing $b_{i,j}$ to 1/0, P_i ignores any further message for the record of P_j.

– *When P_i receives a message $\big(V_{k,j}, \sigma_{\mathsf{sk}_k}(V_{k,j})\big)$ signed by peer P_k for some peer P_j different than P_i and P_k, where $V_{k,j} = \big((\textsc{view}, j), ((\textsc{record}, R_{k,j,p}), \sigma_{\mathsf{sk}_j}(\textsc{record}, R_{k,j,p}))\big)$, and the message was never received before*, then it acts as follows: if $R_{k,j,p}$ is formatted as a non-$\perp$ record and $b_{i,j} =$ '?', then it executes verification $\mathsf{vfy}_{\mathsf{pk}_k}(V_{k,j}, \sigma_{\mathsf{sk}_k}(V_{k,j}))$. If $\mathsf{vfy}_{\mathsf{pk}_k}(V_{k,j}, \sigma_{\mathsf{sk}_k}(V_{k,j})) = 1$, then P_i operates according to either of the following two cases:

1. If $\mathsf{vfy}_{\mathsf{pk}_j}\big((\text{RECORD}, R_{i,j,p}), \sigma_{\mathsf{sk}_j}(\text{RECORD}, R_{i,j,p})\big) = 0$ or $\tilde{B}^i_{k,j,p} \neq \perp$, then P_i sets $\tilde{B}_{i,k,p} \leftarrow \perp$, fixes the bit $b_{i,k}$ to 0[1].
2. If $\mathsf{vfy}_{\mathsf{pk}_j}\big((\text{RECORD}, R_{i,j,p}), \sigma_{\mathsf{sk}_j}(\text{RECORD}, R_{i,j,p})\big) = 1$ and $\tilde{B}^i_{k,j,p} = \perp$, then P_i updates $\mathsf{View}_{i,k,p}$ by setting $\tilde{B}^i_{k,j,p} \leftarrow R_{k,j,p}$. and $\mathsf{View}_{i,p}$ as shown below:

> **(C.1).** If for every $k' \in [N_c] \setminus \{i\}$ such that $\tilde{B}^i_{k',j,p} \neq \perp$, it holds that $\tilde{B}^i_{k',j,p} = \tilde{B}^i_{k,j,p} := \tilde{B}^i_{j,p}$ (i.e. all non-$\perp$ records for j agree on some record $\tilde{B}^i_{j,p}$), then it increases the value of $d_{i,j}$ by 1. Moreover, if $d_{i,j} = t_c + 1$, (i.e., there are $t_c + 1$ matching non-$\perp$ records) and $\tilde{B}_{i,j,p} = \perp$, then it updates as $\tilde{B}_{i,j,p} \leftarrow \tilde{B}^i_{j,p}$ and fixes the bit $b_{i,j}$ to 1.
> **(C.2).** If there is a $k' \in [N_c]$ such that $\tilde{B}^i_{k',j,p} \neq \perp$ and $\tilde{B}^i_{k,j,p} \neq \tilde{B}^i_{k',j,p}$, then it updates as $\tilde{B}_{i,j,p} \leftarrow \perp$ and fixes the bit $b_{i,j}$ to 0.

In either case, upon fixing $b_{i,j}$, P_i ignores any further message for P_j's record[2].
– *When its local clock* $\mathsf{Clock}[P_i]$ *reaches* $T_{\mathsf{request},p}$, P_i broadcasts a request message $\big((\text{REQUEST_VIEW}, j), \sigma_{\mathsf{sk}_i}(\text{REQUEST_VIEW}, j)\big)$, for every P_j that it has not yet fixed the opinion bit $b_{i,j}$. This step is executed to ensure that P_i *will eventually fix its opinion bits for all IC peers*. Upon receiving P_i's request, P_k replies with a signature for a response message $\big(W_{k,j}, \sigma_{\mathsf{sk}_k}(W_{k,j})\big)$, where $W_{k,j} := \big((\text{RESPONSE_VIEW}, j), ((\text{RECORD}, R_{k,j,p}), \sigma_{\mathsf{sk}_j}(\text{RECORD}, R_{k,j,p}))\big)$. Note that here $R_{k,j,p}$ may be $\perp$, reflecting the P_k's lack of direct view for P_j's record. For every P_j that P_i has broadcast $\big((\text{REQUEST_VIEW}, j), \sigma_{\mathsf{sk}_i}(\text{REQUEST_VIEW}, j)\big)$, P_i waits until it collects $N_c - t_c - 1$ distinct valid signed responses. During this wait, it ignores any message in a format other than $\big(W_{k,j}, \sigma_{\mathsf{sk}_k}(W_{k,j})\big)$ or $\big((\text{REQUEST_VIEW}, j), \sigma_{\mathsf{sk}_k}(\text{REQUEST_VIEW}, j)\big)$. When $N_c - t_c - 1$ distinct valid responses are received, it parses the collection of the $N_c - t_c - 1$ responses and its current direct view of P_j's record, $\tilde{B}_{i,j,p}$, to update $\tilde{B}_{i,j,p}$ and fix $b_{i,j}$ as follows:

> **(R.1).** If $\tilde{B}_{i,j,p} \neq \perp$, and all responses for non-$\perp$ records are at least t_c and all match $\tilde{B}_{i,j,p}$, then P_i fixes $b_{i,j}$ to 1.
> **(R.2).** If $\tilde{B}_{i,j,p} = \perp$, and all responses for non-$\perp$ records are at least $t_c + 1$ and all refer to the same record denoted as $\tilde{B}^i_{j,p}$, then P_i sets $\tilde{B}_{i,j,p} \leftarrow \tilde{B}^i_{j,p}$ and fixes $b_{i,j}$ to 1.
> **(R.3).** Otherwise, P_i sets $\tilde{B}_{i,j,p} \leftarrow \perp$ and fixes $b_{i,j}$ to 0.

In any case, upon fixing $b_{i,j}$, P_i ignores any further message for P_j's record [3]. At the end of the **Collection** phase, P_i will have fixed $b_{i,j}$ for all $j \in [N_c]$.
■ **Finalization** phase: having fixed $b_{i,1} \dots, b_{i,N_c}$ and updated its direct view $\mathsf{View}_{i,p} := \langle \tilde{B}_{i,1,p}, \dots, \tilde{B}_{i,N_c,p} \rangle$, peer P_i proceeds as follows: for every pair $(p, x) \in \bigcup_{j:\tilde{B}_{i,j,p} \neq \perp} \tilde{B}_{i,j,p}$, P_i defines the set $N_{i,p}(x)$ that denotes the number of IC peers that, according to its view, have included (p, x) in their records. Formally, we write $N_{i,p}(x) := \#\{j \in [N_c] : (p, x) \in \tilde{B}_{i,j,p}\}$. Then, P_i updates its original record $B_{i,p}$ as follows:

> **(F.1).** If $(p, x) \notin B_{i,p}$, but $N_{i,p}(x) \geq t_c + 1$, then it adds (p, x) in $B_{i,p}$.
> **(F.2).** If $(p, x) \in B_{i,p}$, but $N_{i,p}(x) < t_c + 1$, then it removes (p, x) from $B_{i,p}$.

In any other case, $B_{i,p}$ becomes unchanged[4]. As shown in Lemma 3 in [KKL+18], at the end of the **Finalization** phase, all honest peers have included all honestly posted items for which a receipt has been generated in their local records. Then, they advance to the **Publication** phase described below.

[1] Observe that it is safe for P_i to mark P_k as a malicious, since an honest P_k would neither send two non-$\perp$ views for P_j, nor accept an invalid signature from P_j.

[2] The security of DS ascertains P_i that with $1 - \mathsf{negl}(\kappa)$ probability, only if P_j is malicious, two non-equal records can be valid under P_j's verification key. Thus, in case (C.2), P_i can safely fix the bit $b_{i,j}$ to 0.

[3] Since there are $N_c - t_c \geq t_c + 1$ honest peers, P_i will obtain at least $t_c + 1$ all matching non-$\perp$ views for every honest' peers record (including its own). Thus, in case (R.3), P_i can safely fix $b_{i,j}$ to 0 if it receives inconsistent non-$\perp$ views or less than $t_c + 1$ matching non-$\perp$ views for P_j.

[4] In case (F.2), removal is a safe action for P_i, as every honestly posted item for which a receipt has been generated, is stored in the records of at least $N_c - 2t_c \geq t_c + 1$ honest peers during the **Posting** protocol. Thus, $N_{i,p}(x) < t_c + 1$ implies that either (i) (p, x) was maliciously posted, or (ii) a receipt for (p, x) was not generated.

■ **Publication** phase: each peer P_i threshold signs its record $B_{i,p}$, as it has been updated during the **Finalization** phase, by threshold signing each item in $B_{i,p}$ individually. Formally, $\mathsf{ShareSig}(\mathsf{tsk}_i, (p, B_{i,p})) := \bigcup_{(p,x)\in B_{i,p}} \mathsf{ShareSig}(\mathsf{tsk}_i, (p, x))$. Then, P_i broadcasts the message $\big((p, B_{i,p}), \mathsf{ShareSig}(\mathsf{tsk}_i, (p, B_{i,p}))\big)$ to all peers $AB_1, \dots, AB_{N_w}$ of the AB subsystem.

In turn, each peer AB_j, $j \in [N_w]$ receives and records threshold signature shares for posted items. For every item (p, x) that AB_j receives $N_c - t_c$ valid signatures shares $(k, \sigma_k)_{k\in S}$, where S is a subset of $N_c - t_c$ IC peers, it adds (p, x) to its record $B_p[j]$, initialized as empty, and computes a TSS signature on (p, x) as $\mathsf{TSign}(\mathsf{tsk}, (p, x)) \leftarrow \mathsf{Combine}\big(\mathsf{pk}, \mathsf{pk}_1, \dots, \mathsf{pk}_{N_c}, (p, x), (k, \sigma_k)_{k\in S}\big)$. Upon finalizing $B_p[j]$, AB_j executes the following steps:

1. It sets $\mathsf{TSign}(\mathsf{tsk}, (p, B_p[j])) := \bigcup_{(p,x)\in B_p[j]} \mathsf{TSign}(\mathsf{tsk}, (p, x))$ and when its local clock $\mathsf{Clock}[AB_j]$ reaches $T_{\mathsf{publish},p}$, it publishes the signed record
$$\mathsf{ABreceipt}[p, B_p[j]] := \big((p, B_p[j]), \mathsf{TSign}(\mathsf{tsk}, (p, B_p[j]))\big) .$$
2. By the time that the period p ends (i.e., $\mathsf{Clock}[AB_j] = T_{\mathsf{end},p}$), for $k \in [N_w] \setminus \{j\}$, it performs a read operation on AB_k and reads its record for period p denoted by $B_p[j, k]$ (possibly empty). Then, it publishes the hash $H_\kappa\big(B_p[j, k]\big)$ of the read record.

The VerifyPub algorithm. Let $\mathsf{Prec}[p]$ be the set of all periods preceding p. The total view of AB_j at some moment T during period p, denoted by $L_{\mathsf{pub},j,T}$, is the union of the published BB records $B_{\tilde{p}}[j]$ for all periods $\tilde{p} \in \mathsf{Prec}[p]$.

On input $\big(\langle L_{\mathsf{pub},j,T}\rangle_{j\in[N_w]}, \mathsf{params}\big)$, the algorithm $\mathsf{VerifyPub}$ outputs accept iff for every $j \in [N_w]$ and every $\tilde{p} \in \mathsf{Prec}[p]$ the following hold:

(a). More than $N_w/2$ AB peers that agree on the consistency of the data that AB_j publishes (including AB_j). Formally, there is a subset $\mathcal{I}_j \subseteq [N_w]$ such that $|\mathcal{I}_j| > N_w/2 \quad \text{and} \quad \forall k \in \mathcal{I}_j \setminus \{j\} : H_\kappa\big(B_{\tilde{p}}[k, j]\big) = H_\kappa\big(B_{\tilde{p}}[j]\big)$.

(b). For every $(\tilde{p}, x) \in B_{\tilde{p}}[j]$, it holds that $\mathsf{TVf}\big(\mathsf{pk}, (\tilde{p}, x), \mathsf{TSign}(\mathsf{tsk}, (\tilde{p}, x))\big) = 1$.

An item belongs in the published data of the whole AB system by moment T, denoted by $L_{\mathsf{pub},T}$, if it appears on more than half of the AB peers. Formally,

$$L_{\mathsf{pub},T} := \bigcup_{\tilde{p}\in\mathsf{Prec}[p]} \Big\{(\tilde{p}, x)\Big|\#\{j \in [N_w] : (\tilde{p}, x) \in B_{\tilde{p}}[j]\} > N_w/2\Big\} .$$

Complexity of the new Publishing protocol. Our protocol has a constant number of rounds per period, where the size of transmitted messages is equal to the signature on records of items posted on the said period. In particular, the **Collection** phase has cubic ($\sim (N_c)^3$) communication complexity (the IC peers exchange their views), while the **Publication** phase has quadratic ($\sim N_c \cdot N_w$) communication complexity (the IC peers broadcast their updated records to the AB peers). Overall, the complexity of the new **Publishing** protocol matches the one of the original CS system, as in general, the Floodset algorithm must run in $N_c - t_c + 1$ rounds, where in each round a full quadratic communication for mutual information exchange is required.

Chapter 5

Asymmetric Distributed Trust

5.1 Introduction

Byzantine quorum systems [MR98] are a fundamental primitive for building resilient distributed systems from untrusted components. Given a set of nodes, a quorum system captures a trust assumption on the nodes in terms of potentially malicious protocol participants and colluding groups of nodes. Based on quorum systems, many well-known algorithms for *reliable broadcast*, *shared memory*, *consensus* and more have been implemented; these are the main abstractions to synchronize the correct nodes with each other and to achieve consistency despite the actions of the faulty, so-called *Byzantine* nodes.

Traditionally, trust in a Byzantine quorum system for a set of processes $\mathcal{P}$ has been *symmetric*. In other words, a global assumption specifies which processes may fail, such as the simple and prominent *threshold quorum* assumption, in which any subset of $\mathcal{P}$ of a given maximum size may collude and act against the protocol. The most basic threshold Byzantine quorum system, for example, allows all subsets of up to $f < n/3$ processes to fail. Some classic works also model arbitrary, non-threshold symmetric quorum systems [MR98, HM00], but these have not actually been used in practice.

However, trust is inherently subjective. *De gustibus non est disputandum.* Estimating which processes will function correctly and which ones will misbehave may depend on personal taste. A myriad of local choices influences one process' trust in others, especially because there are so many forms of "malicious" behavior. Some processes might not even be aware of all others, yet a process should not depend on unknown third parties in a distributed collaboration. How can one model asymmetric trust in distributed protocols? Can traditional Byzantine quorum systems be extended to subjective failure assumptions? How do the standard protocols generalize to this model?

In this chapter, we answer these questions and introduce models and protocols for asymmetric distributed trust. We formalize *asymmetric quorum systems* for asynchronous protocols, in which every process can make its own assumptions about Byzantine faults of others. We introduce several protocols with asymmetric trust that strictly generalize the existing algorithms, which require common trust.

Our formalization takes up earlier work by Damgård et al. [DDFN07] and starts out with the notion of a fail-prone system that forms the basis of a symmetric Byzantine quorum system. A global fail-prone system for a process set $\mathcal{P}$ contains all maximal subsets of $\mathcal{P}$ that might jointly fail during an execution. In an asymmetric quorum system, every process specifies its *own* fail-prone system and a corresponding set of local quorums. These local quorum systems satisfy a *consistency condition* that ranges across all processes and a local *availability condition*, and generalize symmetric Byzantine quorum system according to Malkhi and Reiter [MR98].

Interest in consensus protocols based on Byzantine quorum systems has surged recently because of their application to permissioned blockchain networks [CV17, ABB+18]. Typically run by a consortium, such distributed ledgers often use *Byzantine-fault tolerant (BFT)* protocols like PBFT [CL02] for consensus that rely on symmetric threshold quorum systems. The Bitcoin blockchain and many other cryptocurrencies, which triggered this development, started from different assumptions and use so-called permissionless protocols, in which every-

one may participate. Those algorithms capture the relative influence of the participants on consensus decisions by an external factor, such as "proof-of-work" or "proof-of-stake."

A middle ground between permissionless blockchains and BFT-based ones has been introduced by the blockchain networks of Ripple (https://ripple.com) and Stellar (https://stellar.org). Their stated model for achieving network-level consensus uses subjective trust in the sense that each process declares a local list of processes that it "trusts" in the protocol.

Consensus in the *Ripple* blockchain (and for the *XRP* cryptocurrency on the *XRP Ledger*) is executed by its validator nodes. Each validator declares a *Unique Node List (UNL)*, which is a "list of transaction validators a given participant believes will not conspire to defraud them;" but on the other hand, "Ripple provides a default and recommended list which we expand based on watching the history of validators operated by Ripple and third parties" [Rip19]. Many questions have therefore been raised about the kind of decentralization offered by the Ripple protocol. This debate has not yet been resolved.

Stellar was created as an evolution of Ripple that shares much of the same design philosophy. The Stellar consensus protocol [Maz16] powers the *Stellar Lumen (XLM)* cryptocurrency and introduces *federated Byzantine quorum systems (FBQS)*; these bear superficial resemblance with our asymmetric quorum systems but differ technically. Stellar's consensus protocol uses *quorum slices*, which are "the subset of a quorum that can convince one particular node of agreement." In an FBQS, "each node chooses its own quorum slices" and "the system-wide quorums result from these decisions by individual nodes" [Ste15]. However, standard Byzantine quorum systems and FBQS are *not* comparable because (1) an FBQS when instantiated with the same trust assumption for all processes does not reduce to a symmetric quorum system and (2) existing protocols do not generalize to FBQS.

Understanding how such ideas of subjective trust, as manifested in the Ripple and Stellar blockchains, relate to traditional quorum systems is the main motivation for this work. Our contributions are as follows:

- We introduce asymmetric Byzantine quorum systems formally in Section 5.4 as an extension of standard Byzantine quorum systems and discuss some of their properties.
- In Section 5.5, we show two implementations of a shared register, using asymmetric Byzantine quorum systems. The register implements single-writer, multi-reader regular semantics, which means that while only a single party can write to the register, multiple readers read the register and their results must be consistent in that after one reader obtains a newer value, no other reader will obtain an older value.
- We examine broadcast primitives in the Byzantine model with asymmetric trust in Section 5.6. In particular, we define and implement Byzantine consistent and reliable broadcast protocols. The latter primitive is related to a "federated voting" protocol used by Stellar consensus [Maz16].

Before presenting the technical contributions, we state our system model in Section 5.2 and discuss related work in more detail in Section 5.3.

5.2 Preliminaries

Processes. We consider a system of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate with each other. The processes interact asynchronously with each other through exchanging messages. The system itself is asynchronous, i.e., the delivery of messages among processes may be delayed arbitrarily and the processes have no synchronized clocks. Every process is identified by a name, but such identifiers are not made explicit. A protocol for $\mathcal{P}$ consists of a collection of programs with instructions for all processes. Protocols are presented in a modular way using the event-based notation of Cachin et al. [CGR11].

Functionalities. A *functionality* is an abstraction of a distributed computation, either a primitive that may be used by the processes or a service that they will provide. Every functionality in the system is specified through

its *interface*, containing the events that it exposes to protocol implementations that may call it, and its *properties*, which define its behavior. A process may react to a received event by changing their state and triggering further events.

There are two kinds of events in an interface: *input events* that the functionality receives from other abstractions, typically to invoke its services, and *output events*, through which the functionality delivers information or signals a condition a process. The behavior of a functionality is usually stated through a number of properties or through a sequential implementation.

We assume there is a low-level functionality for sending messages over point-to-point links between each pair of processes. In a protocol, this functionality is accessed through the events of "sending a message" and "receiving a message". Point-to-point messages are authenticated, delivered reliably, and output in first-in-first-out (FIFO) order among processes [HT93, CGR11].

Executions and faults. An *execution* starts with all processes in a special initial state; subsequently the processes repeatedly trigger events, react to events, and change their state through computation steps. Every execution is *fair* in the sense that, informally, processes do not halt prematurely when there are still steps to be taken or events to be delivered (see the standard literature for a formal definition [Lyn96]).

A process that follows its protocol during an execution is called *correct*. On the other hand, a *faulty* process may crash or even deviate arbitrarily from its specification, e.g., when *corrupted* by an adversary; such processes are also called *Byzantine*. We consider only Byzantine faults here and assume for simplicity that the faulty processes fail right at the start of an execution.

Idealized digital signatures. A *digital signature scheme* provides two operations, sign_i and verify_i. The invocation of sign_i specifies a process p_i and takes a bit string $m \in \{0,1\}^*$ as input and returns a signature $\sigma \in \{0,1\}^*$ with the response. Only p_i may invoke sign_i. The operation verify_i takes a putative signature σ and a bit string m as parameters and returns a Boolean value with the response. Its implementation satisfies that $\mathit{verify}_i(\sigma, m)$ returns TRUE for any $i \in \{1, \ldots, n\}$ and $m \in \{0,1\}^*$ if and only if p_i has executed $\mathit{sign}_i(m)$ and obtained σ before; otherwise, $\mathit{verify}_i(\sigma, m)$ returns FALSE. Every process may invoke verify.

5.3 Recent Related Work

Damgård et al. [DDFN07] introduce asymmetric trust in the context of synchronous protocols for secure distributed computation by modeling process-specific fail-prone systems. They state the consistency property of asymmetric Byzantine quorums and claim (without proof) that the B^3 property, a specific condition on the size of the fail-prone sets that we describe further in Section 5.4.2, is required for implementing a synchronous broadcast protocol in this setting. However, they do not formalize quorum systems nor discuss asynchronous protocols.

The *Ripple* consensus protocol is run by an open set of validator nodes. The protocol uses votes, similar to standard consensus protocols, whereby each validator only communicates with the validators in its UNL. Each validator chooses its own UNL, which makes it possible for anyone to participate, in principle, similar to proof-of-work blockchains. Early literature suggested that the intersection of the UNLs for every two validators should be at least 20% of each list [SYB14], assuming that also less than one fifth of the validators in the UNL of every node might be faulty. An independent analysis by Armknecht et al. [AKM+15] later argued that this bound must be more than 40%. A recent technical paper of Chase and MacBrough [CM18, Thm. 8] concludes, under the same assumption of $f < n/5$ faulty nodes in every UNL of size n, that the UNL overlap should actually be at least 90%.

However, the same paper also derives a counterexample to the liveness (meaning that the network delivers transactions) of the Ripple consensus protocol [CM18, Sec. 4.2] as soon as two validators don't have "99% UNL overlap." By generalizing the example, this essentially means that the protocol can get stuck *unless all nodes have the same UNL*. According to the widely shared understanding in the field of distributed systems,

though, a protocol needs to satisfy safety ("nothing bad happens", e.g. no two honest nodes have inconsistent views) *and* liveness ("something good happens", e.g. the protocol makes progress) because achieving only one of these properties is trivial. Chase and MacBrough therefore present a devastating verdict on the merit of Ripple's protocol.

The *Stellar consensus protocol (SCP)* also features open membership and lets every node express its own set of trusted nodes [Maz16]. Generalizing from Ripple's flat lists of unique nodes, every node declares a collection of trusted sets called *quorum slices*, whereby a slice is "the subset of a quorum convincing one particular node of agreement." A *quorum* in Stellar is a set of nodes "sufficient to reach agreement," defined as a set of nodes that contains one slice for each member node. The quorum choices of all nodes together yield a *federated Byzantine quorum systems (FBQS)*. The Stellar white paper states properties of FBQS and protocols that build on them. However, these protocols do not map to known protocol primitives in distributed computing. The shortcomings of the Ripple protocol to not apply to Stellar; however, no comprehensive and independent analysis of the Stellar protocol has been published.

García-Pérez and Gotsman [GG18] build a link from FBQS to existing quorum-system concepts by investigating a Byzantine reliable broadcast abstraction in an FBQS. They show that the *federated voting protocol* of Stellar [Maz16] is similar to Bracha's reliable broadcast [Bra87] and that it implements a variation of Byzantine reliable broadcast on an FBQS for executions that contain, additionally, a set of so-called intact nodes.

The paper [GG18], however, uses the FBQS concept of Mazières [Maz16] that is at odds with the usual notion of a Byzantine quorum system in the sense that it does not reduce to a symmetric quorum system for symmetric trust choices. In contrast, we show in the following sections that a natural extension of a symmetric Byzantine quorum system suffices for implementing various protocol primitives with asymmetric trust. We explain, in particular, how to implement the prominent register abstraction with asymmetric trust and investigate multiple broadcast primitives. In particular, Stellar's federated voting protocol and the Byzantine reliable broadcast over an FBQS as described by García-Pérez and Gotsman [GG18], can be shown as straightforward generalizations of Byzantine reliable broadcast with symmetric trust.

A lot of recent work on consensus is in relation with blockchains and other distributed ledger technologies. Two consensus methods that have gained significant visibility there are proof-of-work, which is used by Bitcoin and several other blockchain platforms such as Ethereum, as well as proof-of-stake, which is favored by many newer blockchain platforms such as Cardano or Algorand. On a high-level, the main difference between those systems and asymmetric quorum systems is the type of trust assumptions: In proof-of-work, *every*one assumes that the majority of computing power in the network is controlled by honest parties; in proof-of-stake, *every*one assumes that the majority of stake is controlled by honest parties. By contrast, asymmetric quorum systems allow parties to explicitly specify which other parties they trust; this different type of assumption is suitable for different types of networks. In a nutshell, proof-of-work and proof-of-stake seem suitable for large-scale, "anonymous" networks, whereas asymmetric quorum systems seem more suitable for smaller-scale networks in which the participants know of each other's existence.

5.4 Asymmetric Byzantine Quorum Systems

5.4.1 Symmetric Trust

Quorum systems are well-known in settings with symmetric trust. As demonstrated by many applications to distributed systems, ordinary quorum systems [NW98] and Byzantine quorum systems [MR98] play a crucial role in formulating resilient protocols that tolerate faults through replication [CBPS10]. A quorum system typically ensures a consistency property among the processes in an execution, despite the presence of some faulty processes.

For the model with Byzantine faults, *Byzantine quorum systems* have been introduced by Malkhi and Reiter [MR98]. This notion is defined with respect to a *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$, a collection of subsets of $\mathcal{P}$, none of which is contained in another, such that some $F \in \mathcal{F}$ with $F \subseteq \mathcal{P}$ is called a *fail-prone set* and contains

all processes that may at most fail together in some execution [MR98]. A fail-prone system is the same as the *basis* of an *adversary structure*, which was introduced independently by Hirt and Maurer [HM00].

A fail-prone system captures an assumption on the possible failure patterns that may occur. It specifies all maximal sets of faulty processes that a protocol should tolerate in an execution; this means that a protocol designed for $\mathcal{F}$ achieves its properties as long as the set F of actually faulty processes satisfies $F \in \mathcal{F}$.

Definition 1 (Byzantine quorum system [MR98])**.** A *Byzantine quorum system* for $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where each $Q \in \mathcal{Q}$ is called a *quorum*, such that no quorum is contained in another quorum and the following properties hold:

Consistency: The intersection of any two quorums contains at least one process that is not faulty, i.e.,

$$\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : \; Q_1 \cap Q_2 \not\subseteq F.$$

Availability: For any set of processes that may fail together, there exists a disjoint quorum in $\mathcal{Q}$, i.e.,

$$\forall F \in \mathcal{F} : \; \exists Q \in \mathcal{Q} : \; F \cap Q = \emptyset.$$

The above notion is also known as a *Byzantine dissemination quorum system* [MR98] and allows a protocol to be designed despite arbitrary behavior of the potentially faulty processes. The notion generalizes the usual threshold failure assumption for Byzantine faults [PSL80], which considers that any set of f processes are equally likely to fail.

Similarly to the threshold case, where $n > 3f$ processes overall are needed to tolerate f faulty ones in many Byzantine protocols, Byzantine quorum systems can only exist if not "too many" processes fail.

Definition 2 (Q^3-condition [MR98, HM00])**.** A fail-prone system $\mathcal{F}$ satisfies the *Q^3-condition*, abbreviated as $Q^3(\mathcal{F})$, whenever it holds

$$\forall F_1, F_2, F_3 \in \mathcal{F} : \; \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3.$$

In other words, $Q^3(\mathcal{F})$ means that no *three* fail-prone sets together cover the whole system of processes. A Q^k-condition can be defined like this for any $k \geq 2$ [HM00].

The following lemma considers the *bijective complement* of a process set $\mathcal{S} \subseteq 2^{\mathcal{P}}$, which is defined as $\overline{\mathcal{S}} = \{\mathcal{P} \setminus S | S \in \mathcal{S}\}$, and turns $\mathcal{F}$ into a Byzantine quorum system.

Lemma 1 ([MR98, Theorem 5.4])**.** *Given a fail-prone system $\mathcal{F}$, a Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$. In particular, if $Q^3(\mathcal{F})$ holds, then $\overline{\mathcal{F}}$, the bijective complement of $\mathcal{F}$, is a Byzantine quorum system.*

The quorum system $\mathcal{Q} = \overline{\mathcal{F}}$ is called the *canonical quorum system* of $\mathcal{F}$. According to the duality between $\mathcal{Q}$ and $\mathcal{F}$, properties of $\mathcal{F}$ are often ascribed to $\mathcal{Q}$ as well; for instance, we say $Q^3(\mathcal{Q})$ holds if and only if $Q^3(\mathcal{F})$.

Survivor sets and core sets. Junqueira, Marzullo, and coauthors [JM03, JMHD10] consider generic quorum systems beyond threshold failure models and introduce the notions of a *survivor set* and a *core set*.

A *survivor set* S for $\mathcal{F}$ is a maximal set of processes such that there exists an execution with worst-case failures in which no member of S fails. More formally, $S \subseteq \mathcal{P}$ is a survivor set if and only if there exists $F \in \mathcal{F}$ such that $\mathcal{P} \setminus F = S$. Note that according to our definition of $\mathcal{F}$, every S like this is maximal and satisfies also that for all $S' \subseteq \mathcal{P}$ with $S \subsetneq S'$ and all $F \in \mathcal{F}$ it holds that $\mathcal{P} \setminus F \not\subseteq S'$. For example, under a threshold failure assumption where any f processes may fail, every set of $n - f$ processes is a survivor set. A *survivor set system* $\mathcal{S}$ is the maximal collection of all survivor sets in the sense that no set in $\mathcal{S}$ is contained in another. Observe that the survivor set system $\mathcal{S}$ is the bijective complement of the fail-prone system $\mathcal{F}$; according to Lemma 1, $\mathcal{S}$ is therefore also a Byzantine quorum system.

A *core set* C for $\mathcal{F}$ is a minimal set of processes that contains at least one correct process in every execution. More precisely, $C \subseteq \mathcal{P}$ is a core set whenever (1) for all $F \in \mathcal{F}$, it holds $(\mathcal{P} \setminus F) \cap C \neq \emptyset$ (and, equivalently, $C \not\subseteq F$) and (2) for all $C' \subsetneq C$, there exists $F \in \mathcal{F}$ such that $(\mathcal{P} \setminus F) \cap C' = \emptyset$ (and, equivalently, $C' \subseteq F$). In other words, since $\mathcal{F}$ corresponds directly to $\mathcal{S}$, we can also say that C is a core set if and only if it intersects with every survivor set and is minimal with respect to that property. With the threshold failure assumption, every set of $f+1$ processes is a core set. A *core set system* $\mathcal{C}$ is the minimal collection of all core sets, in the sense that no set in $\mathcal{C}$ is contained in another. Junqueira et al. [JMHD10, Theorem 1] show that survivor sets and core sets are dual and can be characterized in terms of each other. In particular, they show (1) that a core set can also be defined as a process set that intersects with every survivor set and is minimal with respect to that property, and (2) that a survivor set is equivalent to a process set that intersects with every core set and is minimal with respect to that property.

5.4.2 Asymmetric Trust

In our model with asymmetric trust, every process is free to make its own trust assumption and to express this with a fail-prone system. Hence, an *asymmetric fail-prone system* $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ consists of an array of fail-prone systems, where $\mathcal{F}_i$ denotes the trust assumption of p_i. One often assumes $p_i \notin F_i$ for practical reasons, but this is not necessary. This notion has earlier been formalized by Damgård et al. [DDFN07].

For a system $\mathcal{A} \subseteq 2^{\mathcal{P}}$, let $\mathcal{A}^*$ denote the collection of all subsets of the sets in $\mathcal{A}$, that is, $\mathcal{A}^* = \{A' | A' \subseteq A, A \in \mathcal{A}\}$.

Definition 3 (Asymmetric Byzantine quorum system)**.** An *asymmetric Byzantine quorum system* for $\mathbb{F}$ is an array of collections of sets $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$, where $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ for $i \in [1, n]$. The set $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ is called the *quorum system of* p_i and any set $Q_i \in \mathcal{Q}_i$ is called a *quorum (set) for* p_i. It satisfies:

Consistency: The intersection of two quorums for any two processes contains at least one process for which both processes assume that it is not faulty, i.e., for all $i, j \in [1, n]$

$$\forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \ Q_i \cap Q_j \not\subseteq F_{ij}.$$

Availability: For any process p_i and any set of processes that may fail together according to p_i, there exists a disjoint quorum for p_i in $\mathcal{Q}_i$, i.e., for all $i \in [1, n]$

$$\forall F_i \in \mathcal{F}_i : \ \exists Q_i \in \mathcal{Q}_i : \ F_i \cap Q_i = \emptyset.$$

The existence of asymmetric quorum systems can be characterized with a property that generalizes the Q^3-condition for the underlying asymmetric fail-prone systems as follows.

Definition 4 (B^3-condition)**.** An asymmetric fail-prone system $\mathbb{F}$ satisfies the *B^3-condition*, abbreviated as $B^3(\mathbb{F})$, whenever it holds for all $i, j \in [1, n]$ that

$$\forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \ \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$$

The following result is the generalization of Lemma 1 for asymmetric quorum systems; it was stated by Damgård et al. [DDFN07] without proof. As for symmetric quorum systems, we use this result and say that $B^3(\mathbb{Q})$ holds whenever the asymmetric $\mathbb{Q}$ consists of the canonical quorum systems for $\mathbb{F}$ and $B^3(\mathbb{F})$ holds.

Theorem 2. *An asymmetric fail-prone system $\mathbb{F}$ satisfies $B^3(\mathbb{F})$ if and only if there exists an asymmetric quorum system for $\mathbb{F}$.*

Proof. Suppose that $B^3(\mathbb{F})$. We let $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$, where $\mathcal{Q}_i = \overline{\mathcal{F}_i}$ is the canonical quorum system of $\mathcal{F}_i$, and show that $\mathbb{Q}$ is an asymmetric quorum system. Indeed, let $Q_i \in \mathcal{Q}_i$, $Q_j \in \mathcal{Q}_j$, and $F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^*$ for any i and j. Then $F_i = \mathcal{P} \setminus Q_i \in \mathcal{F}_i$ and $F_j = \mathcal{P} \setminus Q_j \in \mathcal{F}_j$ by construction, and therefore, $F_i \cup F_j \cup F_{ij} \neq \mathcal{P}$. This

means there is some $p_k \in \mathcal{P} \setminus (F_i \cup F_j \cup F_{ij})$. This implies in turn that $p_k \in Q_i \cap Q_j$ but $p_k \notin F_{ij}$ and proves the consistency condition. The availability property holds by construction of the canonical quorum systems.

To show the reverse direction, let $\mathbb{Q}$ be a candidate asymmetric Byzantine quorum system for $\mathbb{F}$ that satisfies availability and assume towards a contradiction that $B^3(\mathbb{F})$ does not hold. We show that consistency cannot be fulfilled for $\mathbb{Q}$. By our assumption there are sets F_i, F_j, F_{ij} in $\mathbb{F}$ such that $F_i \cup F_j \cup F_{ij} = \mathcal{P}$, which is the same as $\mathcal{P} \setminus (F_i \cup F_j) \subseteq F_{ij}$. The availability condition for $\mathbb{Q}$ then implies that there are sets $Q_i \in \mathcal{Q}_i$ and $Q_j \in \mathcal{Q}_j$ with $F_i \cap Q_i = \emptyset$ and $F_j \cap Q_j = \emptyset$. Now for every $p_k \in Q_i \cap Q_j$ it holds that $p_k \notin F_i \cup F_j$ by availability and therefore $p_k \in \mathcal{P} \setminus (F_i \cup F_j)$. Taken together this means that $Q_i \cap Q_j \subseteq \mathcal{P} \setminus (F_i \cup F_j) \subseteq F_{ij}$. Hence, $\mathbb{Q}$ does not satisfy the consistency condition and the statement follows. □

Kernels. Given a symmetric Byzantine quorum system $\mathcal{Q}$, we define a *kernel* K as a set of processes that overlaps with every quorum and that is minimal in this respect. Formally, $K \subseteq \mathcal{P}$ is a *kernel of* $\mathcal{Q}$ if and only if

$$\forall Q \in \mathcal{Q} : \ K \cap Q \neq \emptyset$$

and

$$\forall K' \subsetneq K : \exists Q \in \mathcal{Q} : \ K \cap Q = \emptyset.$$

The *kernel system* $\mathcal{K}$ of $\mathcal{Q}$ is the set of all kernels of $\mathcal{Q}$.

For example, under a threshold failure assumption where any f processes may fail and the quorums are all sets of $\lceil \frac{n+f+1}{2} \rceil$ processes, every set of $\lfloor \frac{n-f+1}{2} \rfloor$ processes is a kernel.

The definition of a kernel is related to that of a *core set* by Junqueira et al. [JMHD10], who characterize quorum systems through *survivor sets* instead of fail-prone sets. A core set $\mathcal{C} \subseteq \mathcal{P}$ is a set of parties that intersects with all survivor sets $\mathcal{P} \setminus F$, for $F \in \mathcal{F}$. This means that, given a fail-prone system $\mathcal{F}$, the kernel of the canonical quorum system $\mathcal{Q} = \overline{\mathcal{F}}$ is the same as the core-set system for the fail-prone system $\mathcal{F}$.

Let $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ be an asymmetric fail-prone system and $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$ an asymmetric quorum system for $\mathbb{F}$. An *asymmetric kernel system* for $\mathbb{Q}$ is the array $\mathbb{K} = [\mathcal{K}_1, \ldots, \mathcal{K}_n]$ that consists of the kernel systems for all processes in $\mathcal{P}$; a set $K_i \in \mathcal{K}_i$ is called a *kernel for* p_i.

Asymmetric survivor, core, and double-core sets. Let $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ be an asymmetric fail-prone system. An *asymmetric survivor set system* $\mathbb{S}$ is an array of collections of sets $[\mathcal{S}_1, \ldots, \mathcal{S}_n]$ such that each $\mathcal{S}_i$ is a survivor set system for the fail-prone system $\mathcal{F}_i$. We say that a set $S_i \in \mathcal{S}_i$ is a *survivor set for* p_i.

Analogously, an *asymmetric core set system* $\mathbb{C}$ is an array of collections of sets $[\mathcal{C}_1, \ldots, \mathcal{C}_n]$ such that each $\mathcal{C}_i$ is a core set system for the fail-prone system $\mathcal{F}_i$. We call a set $C_i \in \mathcal{C}_i$ a *core set for* p_i.

Naïve and wise processes. The faults or corruptions occurring in a protocol execution with an underlying quorum system imply a set F of actually *faulty processes*. However, no process knows F and this information is only available to an observer outside the system. With a traditional quorum system $\mathcal{Q}$ designed for a fail-prone set $\mathcal{F}$, the guarantees of a protocol usually hold as long as $F \in \mathcal{F}$. Recall that such protocol properties apply to *correct* processes only but not to faulty ones.

With asymmetric quorums, we further distinguish between two kinds of correct processes, depending on whether they considered F in their trust assumption or not. Given a protocol execution, the processes are therefore partitioned into three types:

Faulty: A process $p_i \in F$ is *faulty*.

Naïve: A correct process p_i for which $F \notin \mathcal{F}_i{}^*$ is called *naïve*.

Wise: A correct process p_i for which $F \in \mathcal{F}_i{}^*$ is called *wise*.

The naïve processes are new for the asymmetric case, as all processes are wise under a symmetric trust assumption. Protocols for asymmetric quorums cannot guarantee the same properties for naïve processes as for wise ones, since the naïve processes may have the "wrong friends."

Guilds. If too many processes are naïve or even fail during a protocol run with asymmetric quorums, then protocol properties cannot be ensured. A *guild* is a set of wise processes that contains at least one quorum for each member; its existence ensures liveness and consistency for typical protocols. This generalizes from protocols for symmetric quorum systems, where the correct processes in every execution form a quorum by definition. (A guild represents a group of influential and well-connected wise processes, like in the real world.)

Definition 5 (Guild)**.** Given a fail-prone system $\mathbb{F}$, an asymmetric quorum system $\mathbb{Q}$ for $\mathbb{F}$, and a protocol execution with faulty processes F, a *guild* $\mathcal{G}$ *for* F satisfies two properties:

Wisdom: $\mathcal{G}$ is a set of wise processes:

$$\forall p_i \in \mathcal{G} : \; F \in {\mathcal{F}_i}^*.$$

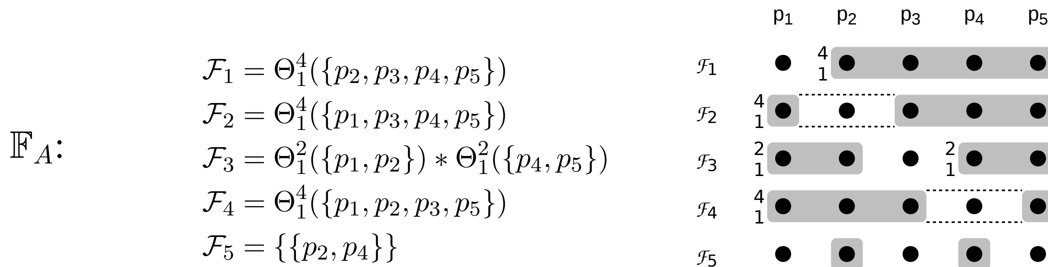
Closure: $\mathcal{G}$ contains a quorum for each of its members:

$$\forall p_i \in \mathcal{G} : \exists Q_i \in \mathcal{Q}_i : \; Q_i \subseteq \mathcal{G}.$$

Superficially a guild seems similar to a "quorum" in the Stellar consensus protocol [Maz16], but the two notions actually differ because a guild contains only wise processes and Stellar's quorums do not distinguish between naïve and wise processes.

Observe that for a specific execution, the union of two guilds is again a guild, since the union consists only of wise processes and contains again a quorum for each member. Hence, every execution with a guild contains a unique *maximal guild* $\mathcal{G}_{\max}$.

Example. We define an example asymmetric fail-prone system $\mathbb{F}_A$ on $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$. The notation $\Theta_k^n(\mathcal{S})$ for a set $\mathcal{S}$ with n elements denotes the "threshold" combination operator and enumerates all subsets of $\mathcal{S}$ of cardinality k. W.l.o.g. every process trusts itself. The diagram below shows fail-prone sets as shaded areas and the notation $\substack{n \\ k}$ in front of a fail-prone set stands for k out of the n processes in the set.



$$\mathbb{F}_A: \quad \begin{aligned} \mathcal{F}_1 &= \Theta_1^4(\{p_2, p_3, p_4, p_5\}) \\ \mathcal{F}_2 &= \Theta_1^4(\{p_1, p_3, p_4, p_5\}) \\ \mathcal{F}_3 &= \Theta_1^2(\{p_1, p_2\}) * \Theta_1^2(\{p_4, p_5\}) \\ \mathcal{F}_4 &= \Theta_1^4(\{p_1, p_2, p_3, p_5\}) \\ \mathcal{F}_5 &= \{\{p_2, p_4\}\} \end{aligned}$$

The operator $*$ for two sets satisfies $\mathcal{A} * \mathcal{B} = \{A \cup B | A \in \mathcal{A}, B \in \mathcal{B}\}$.

As one can verify in a straightforward way, $B^3(\mathbb{F}_A)$ holds. Let $\mathbb{Q}_A$ be the canonical asymmetric quorum system for $\mathbb{F}_A$. Note that since $\mathbb{F}_A$ contains the fail-prone systems of p_3 and p_5 that permit two faulty processes each, this fail-prone system cannot be obtained as a special case of $\Theta_1^5(\{p_1, p_2, p_3, p_4, p_5\})$. When $F = \{p_2, p_4\}$, for example, then processes p_3 and p_5 are wise and p_1 is naïve.

5.5 Shared Memory

This section illustrates a first application of asymmetric quorum systems: how to emulate shared memory, represented by a *register*. Maintaining a shared register reliably in a distributed system subject to faults is perhaps the most fundamental task for which ordinary, symmetric quorum systems have been introduced, in the models with crashes [Gif79] and with Byzantine faults [MR98].

5.5.1 Definitions

Operations and precedence. For the particular *shared-object* functionalities considered here, the processes interact with an object Λ through *operations* provided by Λ. Operations on objects take time and are represented by two events occurring at a process, an *invocation* and a *response*. The *history* of an execution σ consists of the sequence of invocations and responses of Λ occurring in σ. An operation is *complete* in a history if it has a matching response.

An operation o *precedes* another operation o' in a sequence of events σ, denoted $o <_\sigma o'$, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, if $o <_\sigma o'$ then $o <_\pi o'$. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent operations. An execution on a shared object is *well-formed* if the events at each process are alternating invocations and matching responses, starting with an invocation.

Semantics. A *register* with domain $\mathcal{X}$ provides two operations: $\mathit{write}(x)$, which is parameterized by a value $x \in \mathcal{X}$ and outputs a token ACK when it completes; and *read*, which takes no parameter for invocation but outputs a value $x \in \mathcal{X}$ upon completion.

We consider a *single-writer* (or *SW*) register, where only a designated process $p_w \in \mathcal{P}$ may invoke *write*, and permit *multiple readers* (or *MR*), that is, every process may execute a *read* operation. The register is initialized with a special value x_0, which is written by an imaginary *write* operation that occurs before any process invokes operations. We consider *regular* semantics under concurrent access [Lam86]; the extension to other forms of concurrent memory, including an atomic register, proceeds analogously.

It is customary in the literature to assume that the writer and reader processes are correct; with asymmetric quorums we assume explicitly that readers and writers are *wise*. We illustrate below why one cannot extend the guarantees of the register to naïve processes.

Definition 6 (Asymmetric Byzantine SWMR regular register)**.** A protocol emulating an *asymmetric SWMR regular register* satisfies:

Liveness: If a wise process p invokes an operation on the register, p eventually completes the operation.

Safety: Every *read* operation of a wise process that is not concurrent with a *write* returns the value written by the most recent, preceding *write* of a wise process; furthermore, a *read* operation of a wise process concurrent with a *write* of a wise process may also return the value that is written concurrently.

5.5.2 Protocol with Authenticated Data

In Algorithm 1, we describe a protocol for emulating a regular SWMR register with an asymmetric Byzantine quorum system, for a designated writer p_w and a reader $p_r \in \mathcal{P}$. The protocol uses *data authentication* implemented with digital signatures. This protocol is the same as the classic one of Malkhi and Reiter [MR98] that uses a Byzantine dissemination quorum system and where processes send messages to each other over point-to-point links. The difference lies in the individual choices of quorums by the processes and that it ensures safety and liveness for wise processes.

In the register emulation, the writer p_w obtains ACK messages from all processes in a quorum $Q_w \in \mathcal{Q}_w$; likewise, the reader p_r waits for a VALUE message carrying a value/timestamp pair from every process in a quorum $Q_r \in \mathcal{Q}_r$ of the reader.

The function $\mathit{highestval}(S)$ takes a set of timestamp/value pairs S as input and outputs the value in the pair with the largest timestamp, i.e., v such that $(ts, v) \in S$ and $\forall (ts', v') \in S : ts' < ts \vee (ts', v') = (ts, v)$. Note that this v is unique in Algorithm 1 because p_w is correct. The protocol uses digital signatures, modeled by operations sign_i and verify_i, as introduced earlier.

Algorithm 1 Emulation of an asymmetric SWMR regular register (process p_i).

State

wts: sequence number of write operations, stored only by writer p_w
rid: identifier of read operations, used only by reader
ts, v, σ: current state stored by p_i: timestamp, value, signature

upon invocation $write(v)$ **do** // only if p_i is writer p_w
$wts \leftarrow wts + 1$
$\sigma \leftarrow_{\$} sign_w(\text{WRITE} \| w \| wts \| v)$
send message $[\text{WRITE}, wts, v, \sigma]$ to all $p_j \in \mathcal{P}$
wait for receiving a message $[\text{ACK}]$ from all processes in some quorum $Q_w \in \mathcal{Q}_w$

upon invocation *read* **do** // only if p_i is reader p_r
$rid \leftarrow rid + 1$
send message $[\text{READ}, rid]$ to all $p_j \in \mathcal{P}$
wait for receiving messages $[\text{VALUE}, r_j, ts_j, v_j, \sigma_j]$ from all processes in some $Q_r \in \mathcal{Q}_r$ **such that**
$r_j = rid$ **and** $verify_w(\sigma_j, \text{WRITE} \| w \| ts \| v_j)$
return $highestval(\{(ts_j, v_j) | j \in Q_r\})$

upon receiving a message $[\text{WRITE}, ts', v', \sigma']$ from p_w **do** // every process
if $ts' > ts$ **then**
$(ts, v, \sigma) \leftarrow (ts', v', \sigma')$
send message $[\text{ACK}]$ to p_w

upon receiving a message $[\text{READ}, r]$ from p_r **do** // every process
send message $[\text{VALUE}, r, ts, v, \sigma]$ to p_r

Theorem 3. *Algorithm 1 emulates an asymmetric Byzantine SWMR regular register.*

Proof. First we show liveness for wise writer p_w and reader p_r, respectively. Since p_w is wise by assumption, $F \in \mathcal{F}_w{}^*$, and by the availability condition of the quorum system there is $Q_w \in \mathcal{Q}_w$ with $F \cap Q_w = \emptyset$. Therefore, the writer will receive sufficiently many $[\text{ACK}]$ messages and the *write* will return. As p_r is wise, $F \in \mathcal{F}_r{}^*$, and by the analogous condition, there is $Q_r \in \mathcal{Q}_r$ with $F \cap Q_r = \emptyset$. Because p_w is correct and by the properties of the signature scheme, all responses from processes $p_j \in Q_r$ satisfy the checks and *read* returns.

Regarding safety, it is easy to observe that any value output by *read* has been written in some preceding or concurrent *write* operation, and this even holds for naïve readers and writers. This follows from the properties of the signature scheme; *read* verifies the signature and outputs only values with a valid signature produced by p_w.

We now argue that when both the writer and the reader are wise, then *read* outputs a value of either the last preceding *write* or a concurrent *write* and the protocol satisfies safety for a regular register. On a high level, note that $F \in \mathcal{F}_w{}^* \cap \mathcal{F}_r{}^*$ since both are wise. So if p_w writes to a quorum $Q_w \in \mathcal{Q}_w$ and p_r reads from a quorum $Q_r \in \mathcal{Q}_r$, then by consistency of the quorum system $Q_w \cap Q_r \not\subseteq F$ because p_w and p_r are wise. Hence, there is some correct $p_i \in Q_w \cap Q_r$ that received the most recently written value from p_w and returns it to p_r. □

Example. We show why the guarantees of this protocol with asymmetric quorums hold only for wise readers and writers. Consider $\mathbb{Q}_A$ from the last section and an execution in which p_2 and p_4 are faulty, and therefore p_1 is naïve and p_3 and p_5 are wise. A quorum for p_1 consists of p_1 and three processes in $\{p_2, \ldots, p_5\}$; moreover, every process set that contains p_3, one of $\{p_1, p_2\}$ and one of $\{p_4, p_5\}$ is a quorum for p_3.

We illustrate that if naïve p_1 writes, then a wise reader p_3 may violate safety. Suppose that all correct processes, especially p_3, store timestamp/value/signature triples from an operation that has terminated and that wrote x. When p_1 invokes $write(u)$, it obtains $[\text{ACK}]$ messages from all processes except p_3. This is a quorum for p_1. Then p_3 runs a *read* operation and receives the outdated values representing x from itself (p_3 is correct

but has not been involved in writing u) and also from the faulty p_2 and p_4. Hence, p_3 outputs x instead of u.

Analogously, with the same setup of every process initially storing a representation of x but with wise p_3 as writer, suppose p_3 executes *write*(u). It obtains $[\text{ACK}]$ messages from p_2, p_3, and p_4 and terminates. When p_1 subsequently invokes *read* and receives values representing x, from correct p_1 and p_5 and from faulty p_2 and p_4, then p_1 outputs x instead of y and violates safety as a naïve reader.

Since the sample operations are not concurrent, the implication actually holds also for registers with only safe semantics.

5.5.3 Double-Write Protocol without Data Authentication

This section describes a second protocol emulating an asymmetric Byzantine SWMR regular register. In contrast to the previous protocol, it does not use digital signatures for authenticating the data to the reader. Our algorithm generalizes the construction of Abraham et al. [ACKM06] and also assumes a only finite number of write operations occur (*FW-termination*). Furthermore, this algorithm illustrates the use of asymmetric core set systems in the context of an asymmetric-trust protocol. In a nutshell, the difference to the setting in Section 5.5.2 is that, without digital signatures, a reader has no direct way of telling whether a particular message returned by a replica was actually written by the honest writer. Therefore, the reader requires that at least a core set of replicas (i.e. a set including at least one honest replica) returns consistent values.

Theorem 4. *Algorithm 2 emulates an asymmetric Byzantine SWMR regular register, provided there are only finitely many write operations.*

Proof. We first establish safety when the writer p_w and the reader p_r are wise. In that case, $F \in \mathcal{F}_w{}^* \cap \mathcal{F}_r{}^*$. During in a *write* operation, p_w has received PREACK and ACK messages from $Q_w \in \mathcal{Q}_i$ and $Q'_w \in \mathcal{Q}_i$, respectively, and for all $Q_r \in \mathcal{Q}_r$ it holds that $Q_w \cap Q_r \not\subseteq F$ and $Q'_w \cap Q_r \not\subseteq F$.

We now argue that any pair (ts^*, v^*) returned by p_r was written by p_w either in a preceding or a concurrent *write*. From the condition on the core set C_r and (ts^*, v^*) it follows that at least one correct process exists in C_r that stores (ts^*, v^*) as a pre-written or as a written value. Thus, the pair was written by p_w before.

Next we argue that for every completed *write*(v^*) operation, in which p_w has sent $[\text{WRITE}, wts, v^*]$, and for any subsequent *read* operation that selects (ts^*, v^*) and returns v^*, it must hold $wts \leq ts^*$. Namely, the condition on Q_r implies that $ts^* \geq ts_k$ for all $p_k \in Q_r$. By the consistency of the quorum system, it holds that $Q'_w \cap Q_r \not\subseteq F$, so there is a correct process $p_\ell \in Q'_w \cap Q_r$ that has sent ts_ℓ to p_r. Then $ts^* \geq ts_\ell \geq wts$ follows because the timestamp variable of p_ℓ only increases.

The combination of the above two paragraphs implies that for *read* operations that are not concurrent with any *write*, the pair (ts^*, v^*) chosen by *read* was actually written in the immediately preceding *write*. If the *read* operation occurs concurrently with a *write*, then the pair (ts^*, v^*) chosen by *read* may also originate from the concurrent *write*. This establishes the safety property of the SWMR regular register.

We now show liveness. First, if p_w is wise, then there exists a quorum $Q_w \in \mathcal{Q}_w$ such that $Q_w \cap F = \emptyset$. Second, any correct process will eventually receive all $[\text{PREWRITE}, wts, v]$ and $[\text{WRITE}, wts, v]$ messages sent by p_w and process them in the correct order by the assumption of FIFO links. This means that p_w will receive $[\text{PREACK}]$ and $[\text{ACK}]$ messages, respectively, from all processes in one of its quorums, since at least the processes in Q_w will eventually send those.

Liveness for the reader p_r is shown under the condition that p_r is wise and that the *read* operation is concurrent with only finitely many *write* operations. The latter condition implies that there is one last *write* operation that is initiated, but does not necessarily terminate, while *read* is active.

By the assumption that p_w is correct and because messages are received in FIFO order, all messages of that last *write* operation will eventually arrive at the correct processes. Notice also that p_r simply repeats its steps until it succeeds and returns a value that fulfills the condition. Hence, there is a time after which all correct processes reply with VALUE messages that contain pre-written and written timestamp/value pairs from that last operation. It is easy to see that there exist a core set and a quorum for p_r that satisfy the condition and the reader

Algorithm 2 Double-write emulation of an asymmetric SWMR regular register (process p_i).

State
wts: sequence number of write operations, stored only by writer p_w
rid: identifier of read operations, used only by reader
pts, pv, ts, v: current state stored by p_i: pre-written timestamp and value, written timestamp and value

upon invocation *write*(v) **do** // only if p_i is writer p_w
$wts \leftarrow wts + 1$
send message $[\text{PREWRITE}, wts, v]$ to all $p_j \in \mathcal{P}$
wait for receiving a message [PREACK] from all processes in some quorum $Q_w \in \mathcal{Q}_w$
send message $[\text{WRITE}, wts, v]$ to all $p_j \in \mathcal{P}$
wait for receiving a message [ACK] from all processes in some quorum $Q_w \in \mathcal{Q}_w$

upon invocation *read* **do** // only if p_i is reader p_r
$rid \leftarrow rid + 1$
send message $[\text{READ}, rid]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{VALUE}, r_j, pts_j, pv_j, ts_j, v_j]$ from p_j **such that** // only if p_i is reader p_r
$r_j = rid \wedge \big(pts_p = ts_p + 1 \vee (pts, pv) = (ts, v)\big)$ **do**
$readlist[j] \leftarrow (pts_j, pv_j, ts_j, v_j)$
if there exist ts^*, v^*, a core set $C_r \in \mathcal{C}_r$ for p_r, and a quorum $Q_r \in \mathcal{Q}_r$ for p_r **such that**
$C_r \subseteq \big\{p_k | readlist[k] = (pts_k, pv_k, ts_k, v_k) \wedge \big((pts_k, pv_k) = (ts^*, v^*) \vee (ts_k, v_k) = (ts^*, v^*)\big)\big\}$ **and**
$Q_r = \big\{p_k | readlist[k] = (pts_k, pv_k, ts_k, v_k)$
$\wedge \big((ts_k < ts^*) \vee (pts_k, pv_k) = (ts^*, v^*) \vee (ts_k, v_k) = (ts^*, v^*)\big)\big\}$ **then**
return v^*
else
send message $[\text{READ}, rid]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{PREWRITE}, ts', v']$ from p_w **such that** $ts' = pts + 1 \wedge pts = ts$ **do**
$(pts, pv) \leftarrow (ts', v')$
send message [PREACK] to p_w

upon receiving a message $[\text{WRITE}, ts', v']$ from p_w **such that** $ts' = pts \wedge v' = pv$ **do**
$(ts, v) \leftarrow (ts', v')$
send message [ACK] to p_w

upon receiving a message $[\text{READ}, r]$ from p_r **do**
send message $[\text{VALUE}, r, pts, pv, ts, v]$ to p_r

returns. In conclusion, the algorithm emulates an asymmetric regular SWMR register, where liveness holds only for finitely many write operations. □

5.6 Broadcast

This section shows how to implement two *broadcast primitives* tolerating Byzantine faults with asymmetric quorums. Recall from the standard literature [HT93, CBPS10, CGR11] that reliable broadcasts offer basic forms of reliable message delivery and consistency, but they do not impose a total order on delivered messages (as this is equivalent to consensus). The Byzantine broadcast primitives described here, *consistent broadcast* and *reliable broadcast*, are prominent building blocks for many more advanced protocols.

With both primitives, the sender process may broadcast a message m by invoking $\textit{broadcast}(m)$; the broadcast abstraction outputs m to the local application on the process through a $\textit{deliver}(m)$ event. Moreover, the notions of broadcast considered in this section are intended to deliver only one message per instance. Every instance has a distinct (implicit) label and a designated sender p_s. With standard multiplexing techniques one can extend this to a protocol in which all processes may broadcast messages repeatedly [CGR11].

Byzantine consistent broadcast. The simplest such primitive, which has been called *(Byzantine) consistent broadcast* [CGR11], ensures only that those correct processes which deliver a message agree on the content of the message, but they may not agree on termination. In other words, the primitive does not enforce "reliability" such that a correct process outputs a message if and only if all other correct processes produce an output. The events in its interface are denoted by *c-broadcast* and *c-deliver*: the broadcast is initiated by the sender issuing the event *c-broadcast*, and all receivers (upon successful operation) output the event *c-deliver*. (For more details, see Algorithm 3.)

The change of the definition towards asymmetric quorums affects most of its guarantees, which hold only for wise processes but not for all correct ones. This is similar to the definition of a register in Section 5.5.

Definition 7 (Asymmetric Byzantine consistent broadcast)**.** A protocol for *asymmetric (Byzantine) consistent broadcast* satisfies:

Validity: If a correct process p_s *c-broadcasts* a message m, then all wise processes eventually *c-deliver* m.

Consistency: If some wise process *c-delivers* m and another wise process *c-delivers* m', then $m = m'$.

Integrity: For any message m, every correct process *c-delivers* m at most once. Moreover, if the sender p_s is correct and the receiver is wise, then m was previously *c-broadcast* by p_s.

The following protocol is an extension of "authenticated echo broadcast" [CGR11], which goes back to Srikanth and Toueg [ST87]. It is a building block found in many Byzantine fault-tolerant protocols with greater complexity. The adaptation for asymmetric quorums is straightforward: Every process considers its own quorums before *c-delivering* the message.

Theorem 5. *Algorithm 3 implements asymmetric Byzantine consistent broadcast.*

Proof. For the *validity* property, it is straightforward to see that every correct process sends $[\text{ECHO}, m]$. According to the availability condition for the quorum system $\mathcal{Q}_i$ of every wise process p_i and because $F \subseteq F_i$ for some $F_i \in \mathcal{F}_i$, there exists some quorum Q_i for p_i of correct processes that echo m to p_i. Hence, p_i *c-delivers* m.

To show *consistency*, suppose that some wise process p_i has *c-delivered* m_i because of $[\text{ECHO}, m_i]$ messages from a quorum Q_i and another wise p_j has received $[\text{ECHO}, m_j]$ from all processes in $Q_j \in \mathcal{Q}_j$. By the consistency property of $\mathbb{Q}$ it holds $Q_i \cap Q_j \not\subseteq F$; let p_k be this process in $Q_i \cap Q_j$ that is not in F. Because p_k is correct, p_i and p_j received the same message from p_k and $m_i = m_j$.

Algorithm 3 Asymmetric Byzantine consistent broadcast protocol with sender p_s (process p_i)

State

sentecho $\leftarrow$ FALSE: indicates whether p_i has sent ECHO
echos $\leftarrow [\bot]^N$: collects the received ECHO messages from other processes
delivered $\leftarrow$ FALSE: indicates whether p_i has delivered a message

upon invocation *c-broadcast*(m) **do**
send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

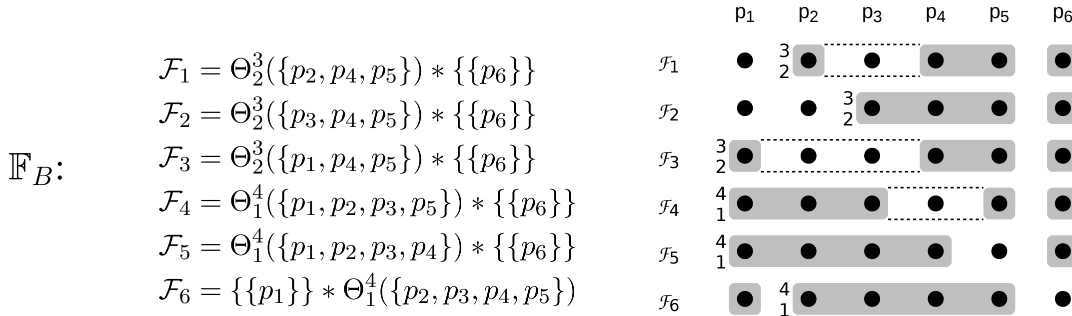
upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
sentecho $\leftarrow$ TRUE
send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
if *echos*$[j] = \bot$ **then**
echos$[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \mathit{echos}[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
delivered $\leftarrow$ TRUE
output *c-deliver*(m)

The first condition of *integrity* is guaranteed by using the *delivered* flag; the second condition holds because because the receiver is wise, and therefore the quorum that it uses for the decision contains some correct processes that have sent $[\text{ECHO}, m]$ with the message m they obtained from p_s according to the protocol. $\square$

Example. We illustrate the broadcast protocols using a six-process asymmetric quorum system $\mathbb{Q}_B$, defined through its fail-prone system $\mathbb{F}_B$. In $\mathbb{F}_B$, as shown below, for p_1, p_2, and p_3, each process always trusts itself, some other process of $\{p_1, p_2, p_3\}$ and one further process in $\{p_1, \ldots, p_5\}$. Process p_4 and p_5 each assumes that at most one other process of $\{p_1, \ldots, p_5\}$ may fail (excluding itself). Moreover, none of the processes $p_1, \ldots, p_5$ ever trusts p_6. For p_6 itself, the fail-prone sets consist of p_1 and one process of $\{p_2, p_3, p_4, p_5\}$.



It is easy to verify that $B^3(\mathbb{F}_B)$ holds; hence, let $\mathbb{Q}_B$ be the canonical quorum system of $\mathbb{F}_B$. Again, there is no reliable process that could be trusted by all and $\mathbb{Q}_B$ is not a special case of a symmetric threshold Byzantine quorum system. With $F = \{p_1, p_5\}$, for instance, processes p_3 and p_6 are wise, p_2 and p_4 are naïve, and there is no guild.

Consider an execution of Algorithm 3 with sender p_4^* and $F = \{p_4^*, p_5^*\}$ (we write p_4^* and p_5^* to denote that they are faulty). This means processes p_1, p_2, p_3 are wise and form a guild because $\{p_1, p_2, p_3\}$ is a quorum for all three; furthermore, p_6 is naïve.

$$
\begin{array}{lll}
 & p_1: \; [\text{ECHO}, x] \rightarrow \mathcal{P} & p_1: \textit{c-deliver}(x) \\
 & p_2: \; [\text{ECHO}, u] \rightarrow \mathcal{P} & p_2: \text{no quorum of } [\text{ECHO}] \text{ in } \mathcal{Q}_2 \\
 & p_3: \; [\text{ECHO}, x] \rightarrow \mathcal{P} & p_3: \text{no quorum of } [\text{ECHO}] \text{ in } \mathcal{Q}_3 \\
p_4^*: \begin{cases} [\text{SEND}, x] \rightarrow p_1, p_3 \\ [\text{SEND}, u] \rightarrow p_2, p_6 \end{cases} & p_4^*: \begin{cases} [\text{ECHO}, x] \rightarrow p_1 \\ [\text{ECHO}, u] \rightarrow p_6 \end{cases} & \\
 & p_5^*: \begin{cases} [\text{ECHO}, x] \rightarrow p_1 \\ [\text{ECHO}, u] \rightarrow p_6 \end{cases} & \\
 & p_6: \; [\text{ECHO}, u] \rightarrow \mathcal{P} & p_6: \textit{c-deliver}(u)
\end{array}
$$

Hence, p_1 receives $[\text{ECHO}, x]$ from $\{p_1, p_3, p_4^*, p_5^*\} \in \mathcal{Q}_1$ and *c-delivers* x, but the other wise processes do not terminate. The naïve p_6 gets $[\text{ECHO}, u]$ from $\{p_2, p_4^*, p_5^*, p_6\} \in \mathcal{Q}_6$ and *c-delivers* $u \neq x$.

Byzantine reliable broadcast. In the symmetric setting, consistent broadcast has been extended to *(Byzantine) reliable broadcast* in a well-known way to address the disagreement about termination among the correct processes [CGR11]. This primitive has the same interface as consistent broadcast, except that its events are called *r-broadcast* and *r-deliver* instead of *c-broadcast* and *c-deliver*, respectively.

A reliable broadcast protocol also has all properties of consistent broadcast, but satisfies the additional *totality* property stated next. Taken together, *consistency* and *totality* imply a notion of *agreement*, similar to what is also ensured by many crash-tolerant broadcast primitives. Analogously to the earlier primitives with asymmetric trust, our notion of an *asymmetric reliable broadcast*, defined next, ensures agreement on termination only for the wise processes, and moreover only for executions with a guild. Also the *validity* of Definition 7 is extended by the assumption of a guild. Intuitively, one needs a guild because the wise processes that make up the guild are self-sufficient, in the sense that the guild contains a quorum of wise processes for each of its members; without that, there may not be enough wise processes.

Definition 8 (Asymmetric Byzantine reliable broadcast)**.** A protocol for *asymmetric (Byzantine) reliable broadcast* is a protocol for asymmetric Byzantine consistent broadcast with the revised *validity* condition and the additional *totality* condition stated next:

Validity: In all executions with a guild, if a correct process p_s *c-broadcasts* a message m, then all processes in the maximal guild eventually *c-deliver* m.

Totality: In all executions with a guild, if a wise process *r-delivers* some message, then all processes in the maximal guild eventually *r-deliver* a message.

The protocol of Bracha [Bra87] implements reliable broadcast subject to Byzantine faults with symmetric trust. It augments the authenticated echo broadcast from Algorithm 3 with a second all-to-all exchange, where each process is supposed to send READY with the payload message that will be *r-delivered*. When a process receives the same m in $2f + 1$ READY messages, in the symmetric model with a threshold Byzantine quorum system, then it *r-delivers* m. Also, a process that receives $[\text{READY}, m]$ from $f + 1$ distinct processes and has not yet sent a READY chimes in and also sends $[\text{READY}, m]$. These two steps ensure totality.

For asymmetric quorums, the conditions of a process p_i receiving $f + 1$ and $2f + 1$ equal READY messages, respectively, generalize to receiving the same message from a kernel for p_i and from a quorum for p_i. Intuitively, the change in the first condition ensures that when a wise process p_i receives the same $[\text{READY}, m]$ message from a kernel for itself, then this kernel intersects with some quorum of wise processes. Therefore, at least one wise process has sent $[\text{READY}, m]$ and p_i can safely adopt m. Furthermore, the change in the second condition relies on the properties of asymmetric quorums to guarantee that whenever some wise process has *r-delivered* m,

Algorithm 4 Asymmetric Byzantine reliable broadcast protocol with sender p_s (process p_i)

State
  sentecho $\leftarrow$ FALSE: indicates whether p_i has sent ECHO
  echos $\leftarrow [\bot]^N$: collects the received ECHO messages from other processes
  sentready $\leftarrow$ FALSE: indicates whether p_i has sent READY
  readys $\leftarrow [\bot]^N$: collects the received READY messages from other processes
  delivered $\leftarrow$ FALSE: indicates whether p_i has delivered a message

upon invocation *r-broadcast*(m) **do**
  send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
  sentecho $\leftarrow$ TRUE
  send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
  if *echos*$[j] = \bot$ **then**
    echos$[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{echos}[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*sentready* **do** // a quorum for p_i
  sentready $\leftarrow$ TRUE
  send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{readys}[j] = m\} \in \mathcal{K}_i$ **and** $\neg$*sentready* **do** // a kernel for p_i
  sentready $\leftarrow$ TRUE
  send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{READY}, m]$ from p_j **do**
  if *readys*$[j] = \bot$ **then**
    readys$[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{readys}[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
  delivered $\leftarrow$ TRUE
  output *r-deliver(m)*

---

then enough correct processes have sent a $[\text{READY}, m]$ message such that all wise processes eventually receive a kernel of $[\text{READY}, m]$ messages and also send $[\text{READY}, m]$.

Applying these changes to Bracha's protocol results in the asymmetric reliable broadcast protocol shown in Algorithm 4. Note that it strictly extends Algorithm 3 by the additional round of READY messages, in the same way as for symmetric trust. For instance, when instantiated with the symmetric threshold quorum system of $n = 3f + 1$ processes, of which f may fail, then every set of $f + 1$ processes is a kernel.

In Algorithm 4, there are two conditions that let a correct p_i send $[\text{READY}, m]$: either when receiving a quorum of $[\text{ECHO}, m]$ messages for itself or after obtaining a kernel for itself of $[\text{READY}, m]$. For the first case, we say p_i *sends* READY *after* ECHO; for the second case, we say p_i *sends* READY *after* READY.

Lemma 6. *In any execution with a guild, there exists a unique m such that whenever a wise process sends a* READY *message, it contains m.*

Proof. Consider first all READY messages sent by wise processes after ECHO. The fact that Algorithm 4 extends Algorithm 3 achieving consistent broadcast, combined with the consistency property in Definition 7 implies immediately that the lemma holds for READY messages sent by wise processes after ECHO.

For the second case, consider the first wise process p_i which sends $[\text{READY}, m']$ after READY. From the protocol it follows that all processes in some kernel $K_i \in \mathcal{K}_i$, which triggered p_i to send $[\text{READY}, m']$, have sent

$[\text{READY}, m']$ to p_i. Moreover, according to the definition of a kernel, K_i overlaps with all quorums for p_i. Since there exists a guild in the execution, at least one of the quorums for p_i consists exclusively of wise processes. Hence, some wise process p_j has sent $[\text{READY}, m']$ to p_i. But since p_i is the first wise process to send READY after READY, it follows that p_j sent $[\text{READY}, m']$ after ECHO; therefore, $m' = m$ from the proof in the first case. Continuing this argument inductively over all READY messages sent after READY by wise processes, in the order these were sent, shows that all those messages contain m and establishes the lemma. □

Theorem 7. *Algorithm 4 implements asymmetric Byzantine reliable broadcast.*

Proof. Recall that the *validity* property assumes there exists a guild $\mathcal{G}$. Since the sender p_s is correct and according to asymmetric quorum availability, every wise process p_i in $\mathcal{G}$ eventually receives a quorum of $[\text{ECHO}, m]$ messages for itself, containing the message m from p_s. According to the protocol, p_i therefore sends $[\text{READY}, m]$ after ECHO unless *sentready* $=$ TRUE; if this is the case, however, p_i has already sent $[\text{READY}, m]$ after READY as ensured by Lemma 6. Hence, every process in $\mathcal{G}$ eventually sends $[\text{READY}, m]$. Then every process in $\mathcal{G}$ receives a quorum for itself of $[\text{READY}, m]$ and *r-delivers* m, as ensured by the properties of a guild and by the protocol.

To establish the *totality* condition, suppose that some wise process p_i has *r-delivered* a message m. Then it has obtained $[\text{READY}, m]$ messages from the processes in some quorum $Q_i \in \mathcal{Q}_i$. Consider any other wise process p_j. Since p_i and p_j are both wise, it holds $F \in \mathcal{F}_i{}^*$ and $F \in \mathcal{F}_j{}^*$, which implies $F \in \mathcal{F}_i{}^* \cap \mathcal{F}_j{}^*$. Then, the set $K = Q_i \setminus F$ intersects every quorum of p_j by quorum consistency and is a kernel for p_j by definition. Since K consists only of correct processes, all of them have sent $[\text{READY}, m]$ also to p_j and p_j eventually sends $[\text{READY}, m]$ as well. This implies that all wise processes eventually send $[\text{READY}, m]$ to all processes. Every process in $\mathcal{G}_{\max}$ therefore receives a quorum for itself of $[\text{READY}, m]$ and *r-delivers* m, as required for totality.

The *consistency* property follows immediately from the preceding argument and from Lemma 6, which implies that all wise processes deliver the same message.

Finally, *integrity* holds because of the *delivered* flag in the protocol and because of the argument showing validity together with Lemma 6. □

Example. Consider again the protocol execution with $\mathbb{Q}_B$ introduced earlier for illustrating asymmetric consistent broadcast. Recall that $F = \{p_4^*, p_5^*\}$, the set $\{p_1, p_2, p_3\}$ is a guild, and p_6 is naïve. The start of the execution is the same as shown previously and omitted. Instead of *c-delivering* x and u, respectively, p_1 and p_6 send $[\text{READY}, x]$ and $[\text{READY}, u]$ to all processes:

...	$p_1 : [\text{READY}, x] \rightarrow \mathcal{P}$			p_1 : *r-deliver*(x)
...	p_2 : no quorum	$p_2 : [\text{READY}, x] \rightarrow \mathcal{P}$		p_2 : *r-deliver*(x)
...	p_3 : no quorum		$p_3 : [\text{READY}, x] \rightarrow \mathcal{P}$	p_3 : *r-deliver*(x)
...	p_4^* : –			
...	p_5^* : –			
...	$p_6 : [\text{READY}, u] \rightarrow \mathcal{P}$	p_6 : no double-core set		

Note that the kernel systems of processes p_1, p_2, and p_3 are $\mathcal{K}_1 = \{\{p_1\}, \{p_3\}\}$, $\mathcal{K}_2 = \{\{p_1\}, \{p_2\}\}$, and $\mathcal{K}_3 = \{\{p_2\}, \{p_3\}\}$. Hence, when p_2 receives $[\text{READY}, x]$ from p_1, it sends $[\text{READY}, x]$ in turn because $\{p_1\}$ is a kernel for p_2, and when p_3 receives this message, then it sends $[\text{READY}, x]$ because $\{p_2\}$ is a kernel for p_3.

Furthermore, since $\{p_1, p_2, p_3\}$ is the maximal guild and contains a quorum for each of its members, all three wise processes *r-deliver* x as implied by *consistency* and *totality*. The naïve p_6 does not *r-deliver* anything, however.

Remarks. Asymmetric reliable broadcast (Definition 8) ensures validity and totality only for processes in the maximal guild. On the other hand, an asymmetric consistent broadcast (Definition 7) ensures validity also for all *wise* processes. We leave it as an open problem to determine whether these guarantees can also be extended to wise processes for asymmetric reliable broadcast and the Bracha protocol. This question is equivalent to determining whether there exist any wise processes outside the maximal guild.

Another open problem concerns the conditions for reacting to READY messages in the asymmetric reliable broadcast protocol. Already in Bracha's protocol for the threshold model [Bra87], a process (1) sends its own READY message upon receiving $f+1$ READY messages and (2) *r-delivers* an output upon receiving $2f+1$ READY messages. These conditions generalize for arbitrary, non-threshold quorum systems to receiving messages (1) from any set that is guaranteed to contain at least one correct process and (2) from any set that still contains at least one process even when any two fail-prone process sets are subtracted. In Algorithm 4, in contrast, a process delivers the payload only after receiving READY messages from one of its quorums. But such a quorum (e.g., $\lceil \frac{n+f+1}{2} \rceil$ processes) may be larger than a set in the second case (e.g., $2f + 1$ processes). It remains interesting to find out whether this discrepancy is necessary.

5.7 Conclusion

The symmetric trust assumption underlying existing protocols in the area of Byzantine fault tolerant protocols does not accurately model the actual trust relationships in reality. In this chapter, we extended the theoretical model for Byzantine quorum systems of Malkhi and Reiter, which models the traditional trust model, to an asymmetric trust model. We initiated the analysis of tasks in this model by providing protocols for shared memory and broadcast, which are based on the traditional protocols from the symmetric setting but adapted to the asymmetric trust assumptions.

Future research will address the design of further protocols in this trust model, especially Byzantine fault tolerant consensus which is needed as a basis of blockchain protocols in the permissioned setting.

Chapter 6

Proof-of-Stake Sidechains

6.1 Introduction

Blockchain protocols and their most prominent application so far, cryptocurrencies like Bitcoin [Nak08], have been gaining increasing popularity and acceptance by a wider community. While enjoying wide adoption, there are several fundamental open questions remaining to be resolved that include (i) Interoperability: How can different blockchains interoperate and exchange assets or other data? (ii) Scalability: How can blockchain protocols scale, especially proportionally to the number of participating nodes? (iii) Upgradability: How can a deployed blockchain protocol codebase evolve to support a new functionality, or correct an implementation problem?

The main function of a blockchain protocol is to organise *application data* into *blocks* so that a set of nodes that evolves over time can arrive eventually to consensus about the sequence of events that took place. The consensus component can be achieved in a number of ways, the most popular is using proof-of-work [DN92] (cf. [Nak08, GKL15]), while a promising alternative is to use proof-of-stake (cf. [Mic16, KRDO17, BPS16, DGKR18]). Application data typically consists of *transactions* indicating some transfer of value as in the case of Bitcoin [Nak08]. The transfer of value can be conditioned on arbitrary predicates called *smart contracts* such as, for example, in Ethereum [But14, Woo14].

The conditions used to validate transactions depend on local blockchain events according to the view of each node and they typically cannot be dependent on other blockchain sessions. Being able to perform operations across blockchains, for instance from a main blockchain such as Bitcoin to a "sidechain" that has some enhanced functionality, has been frequently considered a fundamental technology enabler in the blockchain space.[1]

Sidechains, introduced in [BCD$^+$14], are a way for multiple blockchains to communicate with each other and have one react to events in the other. Sidechains can exist in two forms. In the first case, they are simply a mechanism for two existing *stand-alone blockchains* to communicate, in which case any of the two blockchains can be the sidechain of the other and they are treated as equals. In the second case, the sidechain can be a "child" of an existing blockchain, the *mainchain*, in that its genesis block, the first block of the blockchain, is somehow seeded from the parent blockchain and the child blockchain is meant to depend on the parent blockchain, at least during an initial bootstrapping stage.

A sidechain system can choose to enable certain types of interactions between the participating blockchains. The most basic interaction is the transfer of assets from one blockchain to another. In this application, the nature of the asset transferred is retained in that it is not transformed into a different class of asset (this is in contrast to a related but different concept of *atomic swaps*). As such, it maintains its value and may also be transferred back. The ability to move assets back and forth between two chains is sometimes referred to as a *2-way peg*. Provided the two chains are both secure as individual blockchains, a secure sidechain protocol construction allows this security to be carried on to cross-chain transfers.

A secure sidechain system could be of a great value vis-à-vis all three of the pressing open questions in blockchain systems mentioned above. Specifically:

[1] See e.g., https://blockstream.com/technology/ and [BCD$^+$14].

Interoperability. There are currently hundreds of cryptocurrencies deployed in production. Transferring assets between different chains requires transacting with intermediaries (such as exchanges). Furthermore, there is no way to securely interface with another blockchain to react to events occurring within it. Enabling sidechains allows blockchains of different nature to communicate, including interfacing with the legacy banking system which can be made available through the use of a private ledger.

Scalability. While sidechains were not originally proposed for scalability purposes, they can be used to off-load the load of a blockchain in terms of transactions processed. As long as 2-way pegs are enabled, a particular sidechain can offer specialization by, e.g., industry, in order to avoid requiring the mainchain to handle all the transactions occurring within a particular economic sector. This provides a straightforward way to "shard" blockchains, cf. [LNZ+16, KJG+18, ZMR18].

Upgradability. A child sidechain can be created from a parent mainchain as a means of exploring a new feature, e.g., in the scripting language, or the consensus mechanism without requiring a soft, hard, or velvet fork [KMZ17, ZSJ+18]. The sidechain does not need to maintain its own separate currency, as value can be moved between the sidechain and the mainchain at will. If the feature of the sidechain proves to be popular, the mainchain can eventually be abandoned by moving all assets to the sidechain, which can become the new mainchain.

Given the benefits listed above for distributed ledgers, there is a pressing need to address the question of sidechain security and feasibility, which so far, perhaps surprisingly, has not received any proper formal treatment.

Our contributions. First, we formalize the notion of sidechains by proposing a rigorous cryptographic definition, the first one to the best of our knowledge. The definition is abstract enough to be able to capture the security for blockchains based on proof-of-work, proof-of-stake, and other consensus mechanisms.

A critical security feature of a sidechain system that we formalise is the *firewall property* in which a catastrophic failure in one of the chains, such as a violation of its security assumptions, does not make the other chains vulnerable providing a sense of limited liability.[2] The firewall property formalises and generalises the concept of a blockchain *firewall* which was described in high level in [BCD+14]. Informally the blockchain firewall suggests that no more money can ever return from the sidechain than the amount that was moved into it. Our general firewall property allows relying on an arbitrary definition of exactly how assets can correctly be moved back and forth between the two chains, we capture this by a so-called *validity language*. In case of failure, the firewall ensures that transfers from the sidechain into the mainchain are rejected unless there exists a (not necessarily unique) plausible history of events on the sidechain that could, in case the sidechain was still secure, cause the particular transfers to take place.

Second, we outline a concrete exemplary construction for sidechains for proof-of-stake blockchains. For conciseness our construction is described with respect to a generic PoS blockchain consistent with the Ouroboros protocol [KRDO17] that underlies the Cardano blockchain, which is currently one of the largest pure PoS blockchains by market capitalisation,[3] nevertheless we also discuss how to modify our construction to operate for Ouroboros Praos [DGKR18], Ouroboros Genesis [BGK+18], Snow White [BPS16] and Algorand [Mic16].

We prove our construction secure using standard cryptographic assumptions. We show that our construction (i) supports safe cross-chain value transfers when the security assumptions of both chains are satisfied, namely that a majority of honest stake exists in both chains, and (ii) in case of a one-sided failure, maintains the firewall property, thus containing the damage to the chains whose security conditions have been violated.

[2] To follow the analogy with the term of limited liability in corporate law, a catastrophic sidechain failure is akin to a corporation going bankrupt and unable to pay its debtors. In a similar fashion, a sidechain in which the security assumptions are violated may not be able to cover all of its debtors. We give no assurances regarding assets residing on a sidechain if its security assumptions are broken. However, in the same way that stakeholders of a corporation are personally protected in case of corporate bankruptcy, the mainchain is also protected in case of sidechain security failures. Our security will guarantee that each incoming transaction from a sidechain will always have a valid explanation in the sidechain ledger independently of whether the underlying security assumptions are violated or not. A simple embodiment of this rule is that a sidechain can return to the mainchain at most as many coins as they have been sent to the sidechain over all time.

[3] See https://coinmarketcap.com.

A critical consideration in a sidechain construction is safeguarding a new sidechain in its initial "bootstrapping" stage against a "goldfinger"[4] type of attack [KDF13]. Our construction features a mechanism we call *merged-staking* that allows mainchain stakeholders who have signalled sidechain awareness to create sidechain blocks even without moving stake to the sidechain. In this way, sidechain security can be maintained assuming honest stake majority among the entities that have signaled sidechain awareness that, especially in the bootstrapping stage, are expected to be a large superset of the set of stakeholders that maintain assets in the sidechain.

Our techniques can be used to facilitate various forms of 2-way peggings between two chains. As an illustrative example we focus on a parent-child mainchain-sidechain configuration where sidechain nodes follow also the mainchain (what we call direct observation) while mainchain nodes need to be able to receive cryptographically certified signals from the sidechain maintainers, taking advantage of the proof-of-stake nature of the underlying protocol. This is achieved by having mainchain nodes maintain sufficient information about the sidechain that allows them to authenticate a small subset of sidechain stakeholders that is sufficient to reliably represent the view of a stakeholder majority on the sidechain. This piece of information is updated in regular intervals to account for stake shifting on the sidechain. Exploiting this, each withdrawal transaction from the sidechain to the mainchain is signed by this small subset of sidechain stakeholders. To minimise overheads we batch this authentication information and all the withdrawal transactions from the sidechain in a single message that will be prepared once per "epoch" (in this document we denote each interval with the term epoch). We will refer to this signaling as *cross-chain certification*.

In greater detail, adopting some terminology from [KRDO17], the sidechain certificate is constructed by obtaining signatures from the set of so-called *slot leaders* of the last $\Theta(k)$ slots of the previous epoch, where k is the security parameter. Subsequently, these signatures will be combined together with all necessary information to convince the mainchain nodes (that do not have access to the sidechain) that the sidechain certificate is valid.

We abstract the notion of this trust transition into a new cryptographic primitive called *ad-hoc threshold multisignatures (ATMS)* that we implement in three distinct ways. The first one simply concatenates signatures of elected slot leaders. While secure, the disadvantage of this implementation is that the size of the sidechain certificate is $\Theta(k)$ signatures. An improvement can be achieved by employing multisignatures and Merkle-tree hashing for verification key aggregation; using this we can drop the sidechain-certificate size to $\Theta(r)$ signatures where r slot leaders do not participate in its generation; in the optimistic case $r \ll k$ and thus this scheme can be a significant improvement in practice. Finally, we show that STARKs and bulletproofs [BBHR18, BBB$^+$18] can be used to bring down the size of the certificate to be optimally succinct in the random oracle model. We observe that in the case of an active sidechain (e.g., one that returns assets at least once per epoch) our construction with succinct sidechain certificates has optimal storage requirements in the mainchain.

Related work. Sidechains were first proposed as a high level concept in [BCD$^+$14]. Notable proposed implementations of the concept are given in [Szt15, Ler16]. In these works, no formal proof of security is provided and their performance is sometimes akin to maintaining the whole blockchain within the sidechain, limiting any potential scalability gains. There have been several attempts to create various cross-chain transfer mechanisms including Polkadot [Woo16], Cosmos [Buc16], Blockstream's Liquid [DPW$^+$16] and Interledger [TS]. These constructions differ in various aspects from our work including in that they focus on proof-of-work or private (Byzantine) blockchains, require federations, are not decentralized and — in all cases — lack a formal security model and analysis. Threshold multi-signatures were considered before, e.g., [LHL94], without the ad-hoc characteristic we consider here. A related primitive that has been considered as potentially useful for enabling proof-of-work (PoW) sidechains (rather than PoS ones) is a (non-interactive) proof of proof-of-work [KLS16, KMZ17]; nevertheless, these works do not give a formal security definition for sidechains, nor provide a complete sidechain construction. We reiterate that while we focus on PoS, our definitions and model are fully relevant for the PoW setting as well.

[4]An example of goldfinger attack is one in which the adversary tries to create long forks in order to show that the blockchain is insecure. In this way the adversary can decrease the value of the coins in terms of FIAT currency.

6.2 Preliminaries

6.2.1 Our Model

We employ the model from [DGKR18], which is in turn based on [KRDO17] and [GKL17]. The formalization we use below captures both synchronous and semi-synchronous communication; as well as both semi-adaptive and fully adaptive corruptions.

Protocol Execution

We divide time into discrete units called *slots*. Players are equipped with (roughly) synchronized clocks that indicate the current slot: we assume that any clock drift is subsumed in the slot length. Each slot sl_r is indexed by an integer $r \in \{1, 2, \ldots\}$. We consider a UC-style [Can01] execution of a protocol Π, involving an environment $\mathcal{Z}$, a number of parties P_i, functionalities that these parties can access while running the protocol (such as the $\mathsf{DDiffuse}$ used for communication, described below), and an adversary $\mathcal{A}$. All these entities are interactive algorithms. The environment controls the execution by activating parties via inputs it provides to them. The parties, unless corrupted, respond to such activations by following the protocol Π and invoking the available functionalities as needed.

(Semi-)Adaptive Corruptions.

The adversary influences the protocol execution by interacting with the available functionalities, and by corrupting parties. The adversary can only corrupt a party P_i if it is given permission by the environment $\mathcal{Z}$ running the protocol execution (captured as a special message from $\mathcal{Z}$ to $\mathcal{A}$). Upon receiving permission from the environment, the adversary corrupts P_i after a certain delay of Λ slots, where Λ is a parameter of our model. In particular, if $\Lambda = 0$ we talk about *fully adaptive corruptions* and the corruption is immediate. The model with $\Lambda > 0$ is referred to as allowing Λ-*semi-adaptive corruptions* (as opposed to the *static corruptions model*, where parties can only be corrupted before the start of the execution). A corrupted party P_i will relinquish its entire state to $\mathcal{A}$; from this point on, the adversary will be activated in place of the party P_i.

(Semi-)Synchronous Communication.

We employ the "Delayed Diffuse" functionality $\mathsf{DDiffuse}_\Delta$ given in [DGKR18] to model (semi-)synchronous communication among the parties. It allows each party to diffuse a message once per round, with the guarantee that it will be delivered to all other parties in at most Δ slots (the delay within this interval is under adversarial control). The adversary can also read and reorder all messages that are in transit, as well as inject new messages. We provide a detailed description of the functionality $\mathsf{DDiffuse}_\Delta$ in Section 6.7 for completeness.

We refer to the setting where honest parties communicate via $\mathsf{DDiffuse}_\Delta$ as the Δ-*semi-synchronous setting* and sometimes omit Δ if it is clear from the context. The special case of $\Delta = 0$ is referred to as the *synchronous setting*.

Clearly, the above model is by itself too strong to allow us to prove any meaningful security guarantees for the executed protocol without further restrictions (as it, for example, does not prevent the adversary from corrupting all the participating parties). Therefore, in what follows, we will consider such additional assumptions, and will only provide security guarantees as long as such assumptions are satisfied. These assumptions will be specific to the protocol in consideration, and will be an explicit part of our statements.[5]

6.2.2 Blockchains and Ledgers

A *blockchain* (or a *chain*) (denoted e.g. C) is a sequence of blocks where each one is connected to the previous one by containing its hash.

[5] As an example, we will be assuming that a majority of a certain pool of stake is controlled by uncorrupted parties.

Blockchains (and in general, any sequences) are indexed using bracket notation. $\mathsf{C}[i]$ indicates the ith block, starting from $\mathsf{C}[0]$, the genesis block. $\mathsf{C}[-i]$ indicates the ith block from the end, with $\mathsf{C}[-1]$ being the tip of the blockchain. $\mathsf{C}[i:j]$ indicates a subsequence, or *subchain* of the blockchain starting from block i (inclusive) and ending at block j (exclusive). Any of these two indices can be negative. Omitting one of the two indexes in the range addressing takes the subsequence to the beginning or the end of the blockchain, respectively. Given blocks A and Z in C, we let $\mathsf{C}\{A:Z\}$ denotes the subchain obtained by only keeping the blocks from A (inclusive) to Z (exclusive). Again any of these two blocks can be omitted to indicate a subchain from the beginning or to the end of the blockchain, respectively. In blockchain protocols, each honest party P maintains a currently adopted chain. We denote $\mathsf{C}^P[t]$ the chain adopted by party P at slot t.

A *ledger* (denoted in bold-face, e.g. $\mathbf{L}$) is a mechanism for maintaining a sequence of transactions, often stored in the form of a blockchain. In this document, we slightly abuse the language by letting $\mathbf{L}$ (without further qualifiers) interchangeably refer to the algorithms used to maintain the sequence, and all the views of the participants of the state of these algorithms when being executed. For example, the (existing) ledger Bitcoin consists of the set of all transactions that ever took place in the Bitcoin network, the current Unspent Transaction Output (UTXO) set, as well as the local views of all the participants.

In contrast, we call a *ledger state* a concrete sequence of transactions $\mathsf{tx}_1, \mathsf{tx}_2, \ldots$ stored in the *stable* part of a ledger $\mathbf{L}$, typically as viewed by a particular party. Hence, in every blockchain-based ledger $\mathbf{L}$, every fixed chain C defines a concrete ledger state by applying the interpretation rules given as a part of the description of $\mathbf{L}$ (for example, the ledger state is obtained from the blockchain by dropping the last k blocks and serializing the transactions in the remaining blocks). We maintain the typographic convention that a ledger state (e.g. L) always belongs to the bold-face ledger of the same name (e.g. $\mathbf{L}$). We denote by $\mathbf{L}^P[t]$ the ledger state of a ledger $\mathbf{L}$ as viewed by a party P at the beginning of a time slot t, and by $\check{\mathbf{L}}^P[t]$ the complete state of the ledger (at time t) including all pending transactions that are not stable yet. For two ledger states (or, more generally, any sequences), we denote by $\preceq$ the prefix relation.

Recall the definitions of persistence and liveness of a robust public transaction ledger given in the most recent version of [GKL17]:

Persistence. For any two honest parties P_1, P_2 and two time slots $t_1 \leq t_2$, it holds $\mathbf{L}^{P_1}[t_1] \preceq \check{\mathbf{L}}^{P_2}[t_2]$.

Liveness. If all honest parties in the system attempt to include a transaction tx then, at any slot t after u slots (called the liveness parameter), any honest party P, if queried, will report $\mathsf{tx} \in \mathbf{L}^P[t]$.

For a ledger $\mathbf{L}$ that satisfies persistence at time t, we denote by $\mathbf{L}^{\cup}[t]$ (resp. $\mathbf{L}^{\cap}[t]$) the sequence of transactions that are seen as included in the ledger by *at least one* (resp., *all*) of the honest parties. Finally, $\mathsf{length}(\mathbf{L})$ denotes the length of the ledger $\mathbf{L}$, i.e., the number of transactions it contains.

6.2.3 Underlying Proof-of-Stake Protocols

For conciseness we present our construction on a generic PoS protocol based on Ouroboros PoS [KRDO17]. As we outline in Section 6.8, our construction can be easily adapted to other provably secure proof-of-stake protocols: Ouroboros Praos [DGKR18], Ouroboros Genesis [BGK+18], Snow White [BPS16], and Algorand [Mic16]. While a full understanding of all details of these protocols is not required to follow our work, an overview of Ouroboros is helpful to follow the main body of this part of the document. We provide this high-level overview here, and point an interested reader to Section 6.8 for details on the other protocols.

Ouroboros.

The protocol operates (and was analyzed) in the synchronous model with semi-adaptive corruptions. In each slot, each of the parties can determine whether she qualifies as a so-called *slot leader* for this slot. The event of a particular party becoming a slot leader occurs with a probability proportional to the stake controlled by that party and is independent for two different slots. It is determined by a public, deterministic computation from the

stake distribution and so-called *epoch randomness* (we will discuss shortly where this randomness comes from) in such a way that for each slot, exactly one leader is elected.

If a party is elected to act as a slot leader for the current slot, she is allowed to create, sign, and broadcast a block (containing transactions that move stake among stakeholders). Parties participating in the protocol are collecting such valid blocks and always update their current state to reflect the longest chain they have seen so far that did not fork from their previous state by too many blocks into the past.

Multiple slots are collected into *epochs*, each of which contains $R \in \mathbb{N}$ slots. The security arguments in [KRDO17] require $R \geq 10k$ for a security parameter κ; we will consider $R = 12k$ as additional $2k$ slots in each epoch will be useful for our construction. Each epoch is indexed by an index $j \in \mathbb{N}$. During an epoch j, the stake distribution that is used for slot leader election corresponds to the distribution recorded in the ledger up to a particular slot of epoch $j-1$, chosen in a way that guarantees that by the end of epoch $j-1$, there is consensus on the chain up to this slot. (More concretely, this is the latest slot of epoch $j-1$ that appears in the first $4k$ out of its total R slots.) Additionally, the *epoch randomness* η_j for epoch j is derived during the epoch $j-1$ via a *guaranteed-output delivery coin tossing* protocol that is executed by the epoch slot leaders, and is available after $10k$ slots of epoch $j-1$ have passed (we refer to [KRDO17] for more details).

In our treatment, we will refer to the relevant parts of the above-described protocol as follows:

$\mathsf{GetDistr}(j)$ returns the stake distribution SD_j to be used for epoch j, as recorded in the chain up to slot $4k$ of epoch $j-1$;

$\mathsf{GetRandomness}(j)$ returns the randomness η_j for epoch j as derived during epoch $j-1$;

$\mathsf{ValidateConsensusLevel}(\mathsf{C})$ checks the consensus-level validity of a given chain C: it verifies that all block hashes are correct, signatures are valid and belong to eligible slot leaders;

$\mathsf{PickWinningChain}(\mathsf{C}, \mathcal{C})$ applies the chain-selection rule: from a set of chains $\{\mathsf{C}\} \cup \mathcal{C}$ it chooses the longest one that does not fork from the current chain C more than k blocks in the past;

$\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ determines whether a party U is elected a slot leader for the slot sl of epoch j, given stake distribution SD_j and randomness η_j.

Moreover, the function $\mathsf{EpochIndex}$ (resp. $\mathsf{SlotIndex}$) always returns the index of the current epoch (resp. slot), and the event $\mathsf{NewEpoch}$ (resp. $\mathsf{NewSlot}$) denotes the start of a new epoch (resp. slot). Since we use these functions in a black-box manner, our construction can be readily adapted to PoS protocols with a similar structure that differ in the details of these procedures.

Ouroboros was shown in [KRDO17] to achieve both persistence and liveness under the following assumptions: (1) synchronous communication; (2) $2R$-semi-adaptive corruptions; (3) majority of stake in the stake distribution for each epoch is always controlled by honest parties during that epoch.

6.3 Defining Security of Pegged Ledgers

In this section we give the first formal definition of security desiderata for a system of pegged ledgers (popularly often called sidechains). We start by conveying its intuition and then proceed to the formal treatment.

We consider a setting where a set of parties run a protocol maintaining n ledgers $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_n$, each of the ledgers potentially carrying many different assets. (This protocol might of course be a combination of subprotocols for each of the ledgers.) For each $i \in [n]$, we denote by $\mathbb{A}_i$ the security assumption required by $\mathbf{L}_i$: For example, $\mathbb{A}_i$ may denote that there has never been a majority of hashing power (or stake in a particular asset, on this ledger or elsewhere) under the control of the adversary; that a particular entity (in case of a centralized ledger) was not corrupted; and so on. We assume that all $\mathbb{A}_i$ are *monotone* in the sense that once violated, they cannot become true again. Formally, $\mathbb{A}_i$ is a sequence of events $\mathbb{A}_i[t]$ for each time slot t that satisfy $\neg\mathbb{A}_i[t] \Rightarrow \neg\mathbb{A}_i[t+1]$ for all t.

There is an a priori unlimited number of (types of) assets, each asset representing e.g. a different cryptocurrency. For simplicity we assume that assets of the same type are fungible, but our treatment easily covers also non-fungible assets. We will allow specific rules of behavior for each asset (called *validity languages*), and each asset behaves according to these rules on each of the ledgers where it is present.

We will fix an operator $\mathsf{merge}(\cdot)$ that merges a set of ledger states $\mathcal{L} = \{\mathsf{L}_1, \mathsf{L}_2, \dots, \mathsf{L}_n\}$ into a single ledger state denoted by $\mathsf{merge}(\mathcal{L})$. We will discuss concrete instantiations of $\mathsf{merge}(\cdot)$ later, for now simply assume that some canonical way of merging all ledger states into one is given.

Informally, at any point t during the execution, our security definition only provides guarantees to the subset $\mathcal{S}$ of ledgers that have their security assumptions $\mathbb{A}_i[t]$ satisfied (and hence are all considered uncorrupted). We require that:

- each ledger in $\mathcal{S}$ individually maintains both persistence and liveness;
- for each asset A, when looking at the sequence of all A-transactions σ that occurred on the ledgers in $\mathcal{S}$ (sequentialized via the merge operator), there must exist a hypothetical sequence of A-transactions τ that could have happened on the compromised ledgers, such that the merge of σ and τ would be valid according to the validity language of A.

We now proceed to formalize the above intuition.

Definition 9 (Assets, syntactically valid transactions)**.** For an asset A, we denote by $\mathcal{T}_\mathsf{A}$ the *valid transaction set of* A, i.e., the set of all syntactically valid transactions involving A. For a ledger $\mathbf{L}$ we denote by $\mathcal{T}_\mathbf{L}$ the set of transactions that can be included into $\mathbf{L}$. For notational convenience, we define $\mathcal{T}_{\mathsf{A},\mathbf{L}} \triangleq \mathcal{T}_\mathsf{A} \cap \mathcal{T}_\mathbf{L}$. Let $\mathsf{Assets}(\mathbf{L})$ denote the set of all assets that are supported by $\mathbf{L}$. Formally, $\mathsf{Assets}(\mathbf{L}) \triangleq \{\mathsf{A} : \mathcal{T}_{\mathsf{A},\mathbf{L}} \neq \emptyset\}$.

We assume that each transaction pertains to a particular asset and belongs to a particular ledger, i.e., for distinct $\mathsf{A}_1 \neq \mathsf{A}_2$ and $\mathbf{L}_1 \neq \mathbf{L}_2$, we have that $\mathcal{T}_{\mathsf{A}_1} \cap \mathcal{T}_{\mathsf{A}_2} = \emptyset$ and $\mathcal{T}_{\mathbf{L}_1} \cap \mathcal{T}_{\mathbf{L}_2} = \emptyset$. However, our treatment can be easily generalized to alleviate this restriction.

We now generically characterize the *validity* of a sequence of transactions involving a particular asset. This is captured individually for each asset via a notion of an asset's *validity language*, which is simply a set of words over the alphabet of this asset's transactions. The asset's validity language is meant to capture how the asset is mandated to behave in the system. Let ε denote the empty sequence and $\|$ represent concatenation.

Definition 10 (Asset validity language)**.** For an asset A, the *asset validity language of* A is any language $\mathbb{V}_\mathsf{A} \subseteq \mathcal{T}_\mathsf{A}^*$ that satisfies the following properties:

1. **Base.** $\varepsilon \in \mathbb{V}_\mathsf{A}$.
2. **Monotonicity.** For any $w, w' \in \mathcal{T}_\mathsf{A}^*$ we have $w \notin \mathbb{V}_\mathsf{A} \Rightarrow w \,\|\, w' \notin \mathbb{V}_\mathsf{A}$.
3. **Uniqueness of transactions.** Words from $\mathbb{V}_\mathsf{A}$ never contain the same transaction twice: for any $\mathsf{tx} \in \mathcal{T}_\mathsf{A}$ and any $w_1, w_2, w_3 \in \mathcal{T}_\mathsf{A}^*$ we have $w_1 \,\|\, \mathsf{tx} \,\|\, w_2 \,\|\, \mathsf{tx} \,\|\, w_3 \notin \mathbb{V}_\mathsf{A}$.

The first condition in the definition above is trivial, the second one mandates the natural property that if a sequence of transactions is invalid, it cannot become valid again by adding further transactions. Finally, the third condition reflects a natural "uniqueness" property of transactions in existing implementations. While not necessary for our treatment, it allows for some simplifications.

The following definition allows us to focus on a particular asset or ledger within a sequence of transactions.

Definition 11 (Ledger state projection)**.** Given a ledger state L, we call a *projection of* L *with respect to a set* $\mathcal{X}$ (and denote by $\pi_\mathcal{X}(\mathsf{L})$) the ledger state that is obtained from L by removing all transactions not in $\mathcal{X}$. To simplify notation, we will use π_A and $\pi_\mathcal{I}$ as a shorthand for $\pi_{\mathcal{T}_\mathsf{A}}$ and $\pi_{\bigcup_{i\in\mathcal{I}} \mathcal{T}_{\mathbf{L}_i}}$, denoting the projection of the transactions of a ledger state with respect to particular asset A or a particular set of individual ledger indices. Naturally, for a language $\mathbb{V}$ we define the *projected language* $\pi_\mathcal{X}(\mathbb{V}) := \{\pi_\mathcal{X}(w) : w \in \mathbb{V}\}$, which contains all the sequences of transactions from the original language, each of them projected with respect to $\mathcal{X}$.

The concept of *effect transactions* below captures ledger interoperability at the syntactic level.

Definition 12 (Effect Transactions)**.** For two ledgers $\mathbf{L}$ and $\mathbf{L}'$, the *effect mapping* is a mapping of the form $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}\colon \mathcal{T}_{\mathbf{L}} \to (\mathcal{T}_{\mathbf{L}'} \cup \{\bot\})$. A transaction $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \bot$ is called the *effect transaction* of the transaction tx.

Intuitively, for any transaction $\mathsf{tx} \in \mathcal{T}_{\mathbf{L}}$, the corresponding transaction $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \in \mathcal{T}_{\mathbf{L}'} \cup \{\bot\}$ identifies the necessary effect on ledger $\mathbf{L}'$ of the event of the inclusion of the transaction tx into the ledger $\mathbf{L}$. With foresight, in an implementation of a system of ledgers where a "pegging" exists, the transaction $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx})$ has to be eventually valid and includable in $\mathbf{L}'$ in response to the inclusion of tx in $\mathbf{L}$. Additionally, throughout the document we assume that an effect transaction is always clearly identifiable as such, and its corresponding "sending" transaction can be derived from it; our instantiation does have this property.

We use a special symbol $\bot$ to indicate that the transaction tx does not necessitate any action on $\mathbf{L}'$ (this will be the case for most transactions). We will now be interested mostly in transactions that *do* require an action on the other ledger.

Definition 13 (Cross-Ledger Transfers)**.** For two ledgers $\mathbf{L}$ and $\mathbf{L}'$ and an effect mapping $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\cdot)$, we refer to a transaction in $\mathcal{T}_{\mathbf{L}}$ that requires some effect on $\mathbf{L}'$ as a $(\mathbf{L}, \mathbf{L}')$*-cross-ledger transfer transaction* (or *cross-ledger transfer* for short). The set of all cross-ledger transfers is denoted by $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \subseteq \mathcal{T}_{\mathbf{L}}$, formally $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \triangleq \{\mathsf{tx} \in \mathcal{T}_{\mathbf{L}} \,:\, \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \bot\}$.

Given ledger states $\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n$, we need to consider a joint ordered view of the transactions in all these ledgers. This is provided by the merge operator. Intuitively, merge allows us to create a combined view of multiple ledgers, putting all of the transactions across multiple ledgers into a linear ordering. We expect that even if certain ledgers are missing from its input, merge is still able to produce a global ordering for the remaining ledgers. With foresight, this ability of the merge operator will enable us to reason about the situation when some ledgers fail: In that case, the respective inputs to the merge function will be missing. The merge function definition below depends on the effect mappings, we keep this dependence implicit for simpler notation.

Definition 14 (Merging ledger states)**.** The $\mathsf{merge}(\cdot)$ function is any mapping taking a subset of ledger states $\mathcal{L} \subseteq \{\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n\}$ and producing a ledger state $\mathsf{merge}(\mathcal{L})$ such that:

1. **Partitioning.** The ledger states in $\mathcal{L}$ are disjoint subsequences of $\mathsf{merge}(\mathcal{L})$ that cover the whole sequence $\mathsf{merge}(\mathcal{L})$.
2. **Topological soundness.** For any $i \neq j$ such that $\mathsf{L}_i, \mathsf{L}_j \in \mathcal{L}$ and any two transactions $\mathsf{tx} \in \mathsf{L}_i$ and $\mathsf{tx}' \in \mathsf{L}_j$, if $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx})$ then tx precedes tx' in $\mathsf{merge}(\mathcal{L})$.

We will require that our validity languages are *correct* in the following sense.

Definition 15 (Correctness of $\mathbb{V}_{\mathsf{A}}$)**.** A validity language $\mathbb{V}_{\mathsf{A}}$ is *correct* with respect to a mapping $\mathsf{merge}(\cdot)$, if for any ledger states $\mathcal{L} \triangleq (\mathsf{L}_1, \ldots, \mathsf{L}_n)$ such that $\pi_{\mathsf{A}}(\mathsf{merge}(\mathcal{L})) \in \mathbb{V}_{\mathsf{A}}$, indices $i \neq j$, and any cross-ledger transfer $\mathsf{tx} \in \mathsf{L}_i \cap \mathcal{T}^{\mathsf{cl}}_{\mathbf{L}_i,\mathbf{L}_j}$ such that $\mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx}) = \mathsf{tx}' \neq \bot$ is not in L_j, we have

$$\pi_{\mathsf{A}}\left(\mathsf{merge}\left(\mathsf{L}_1, \ldots, \mathsf{L}_i, \ldots, \mathsf{L}_j \,\|\, \mathsf{tx}', \ldots, \mathsf{L}_n\right)\right) \in \mathbb{V}_{\mathsf{A}} \,.$$

The above definition makes sure that if a cross-ledger transfer of an asset A is included into some ledger $\mathbf{L}_i$ and mandates an effect transaction on $\mathbf{L}_j$, then the inclusion of this effect transaction will be consistent with $\mathbb{V}_{\mathsf{A}}$. Note that this does not yet guarantee that the effect transaction will indeed be included into $\mathbf{L}_j$, this will be provided by the liveness of $\mathbf{L}_j$ required below.

We are now ready to give our main security definition. In what follows, we call a *system-of-ledgers protocol* any protocol run by a (possibly dynamically changing) set of parties that maintains an evolving state of n ledgers $\{\mathbf{L}_i\}_{i\in[n]}$.

Definition 16 (Pegging security)**.** A system-of-ledgers protocol Π for $\{\mathbf{L}_i\}_{i\in[n]}$ is *pegging-secure* with liveness parameter $u \in \mathbb{N}$ with respect to:

- a set of assumptions $\mathbb{A}_i$ for ledgers $\{\mathbf{L}_i\}_{i\in[n]}$,
- a merge mapping $\mathsf{merge}\,(\cdot)$,
- validity languages $\mathbb{V}_\mathsf{A}$ for each $\mathsf{A} \in \bigcup_{i\in[n]} \mathsf{Assets}(\mathbf{L}_i)$,

if for all PPT adversaries, all slots t and for $\mathcal{S}_t \triangleq \{i \ : \ \mathbb{A}_i[t] \text{ holds}\}$ we have that except with negligible probability in the security parameter:

Ledger persistence: For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the persistence property.

Ledger liveness: For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the liveness property parametrized by u.

Firewall: For all $\mathsf{A} \in \bigcup_{i\in\mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$,

$$\pi_\mathsf{A}\left(\mathsf{merge}\left(\left\{\mathbf{L}_i^\cup[t] \ : \ i \in \mathcal{S}_t\right\}\right)\right) \in \pi_{\mathcal{S}_t}(\mathbb{V}_\mathsf{A}) \ .$$

Intuitively, the firewall property above gives the following guarantee: If the security assumption of a particular sidechain has been violated, we demand that the sequence of transactions σ that appears in the still uncompromised ledgers is a valid projection of some word from the asset validity language onto these ledgers. This means that there exists a sequence of transactions τ that *could have happened* on the compromised ledgers, such that it would "justify" the current state of the uncompromised ledgers as a valid state. Of course, we don't know whether this sequence τ actually occurred on the compromised ledger, however, given that this ledger itself no longer provides any reliable state, this is the best guarantee we can still offer to the uncompromised ledgers.

Looking ahead, when we define a particular validity language for our concrete, fungible, constant-supply asset, we will see that this property will translate into the mainchain maintaining "limited liability" towards the sidechain: the amount of money transferred back from the sidechain can never exceed the amount of money that was previously moved towards the sidechain, because no plausible history of sidechain transactions can exist that would justify such a transfer.

6.4 Implementing Pegged Ledgers

We present a construction for pegged ledgers that is based on Ouroboros PoS [KRDO17], but also applicable to other PoS systems such as Snow White [BPS16] and Algorand [Mic16] (for a discussion of such adaptations, see Section 6.8). Our protocol will implement a system of ledgers with pegging security according to Definition 16 under an assumption on the relative stake power of the adversary that will be detailed below.

The main challenge in implementing pegged ledgers is to facilitate secure cross-chain transfers. We consider two approaches to such transfers and refer to them as *direct observation* or *cross-chain certification*. Consider two pegged ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$. Direct observation of $\mathbf{L}_1$ means that every node of $\mathbf{L}_2$ follows and validates $\mathbf{L}_1$; it is easy to see that this enables transfers from $\mathbf{L}_1$ to $\mathbf{L}_2$. On the other hand, cross-chain certification of $\mathbf{L}_2$ means that $\mathbf{L}_1$ contains appropriate cryptographic information sufficient to validate data issued by the nodes following $\mathbf{L}_2$. This allows transfers of assets from $\mathbf{L}_2$, as long as they are certified, to be accepted by $\mathbf{L}_1$-nodes without following $\mathbf{L}_2$. The choice between direct observation and cross-chain certification can be made independently for each direction of transfers between $\mathbf{L}_1$ and $\mathbf{L}_2$, any of the 4 variants is possible (cf. Figure 6.1).

Another aspect of implementing pegged ledgers in the PoS context is the choice of stake distribution that underlies the PoS on each of the chains. We again consider two options, which we call *independent staking* and *merged staking*. In independent staking, blocks on say $\mathbf{L}_1$ are "produced by" coins from $\mathbf{L}_1$ (in other words, the block-creating rights on $\mathbf{L}_1$ are attributed based on the stake distribution recorded on $\mathbf{L}_1$ only). In contrast, with

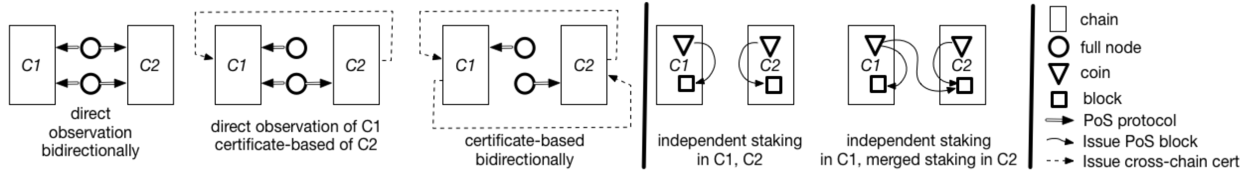


Figure 6.1: Deployment options for PoS Sidechains.

merged staking, blocks on $\mathbf{L}_1$ are produced either by coins on $\mathbf{L}_1$, or coins on $\mathbf{L}_2$ that have, via their staking key, declared support of $\mathbf{L}_1$ (but otherwise remain on $\mathbf{L}_1$); see Figure 6.1. Also here, all 4 combinations are possible.

In our construction we choose an exemplary configuration between two ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$, so that direct observation is applied to $\mathbf{L}_1$, cross-chain certification to $\mathbf{L}_2$, independent-staking in $\mathbf{L}_1$ and merged staking in $\mathbf{L}_2$. As a result, all stakeholders in $\mathbf{L}_2$ also keep track of chain development on $\mathbf{L}_1$ (and hence run a full node for $\mathbf{L}_1$) while the opposite is not necessary, i.e., $\mathbf{L}_1$ stakeholders can be oblivious of transactions and blocks being added to $\mathbf{L}_2$. This illustrates the two basic possibilities of pegging and can be easily adapted to any other of the configurations between two ledgers in Figure 6.1.

In order to reflect the asymmetry between the two chains in our exemplary construction we will refer to $\mathbf{L}_1$ as the "mainchain" $\mathbf{MC}$, and to $\mathbf{L}_2$ as the "sidechain" $\mathbf{SC}$. To elaborate further on this concrete asymmetric use case, we also fully specify how the sidechain can be initialized from scratch, assuming that the mainchain already exists.

The pegging with the sidechain will be provided with respect to a specific asset of $\mathbf{MC}$ that will be created on $\mathbf{MC}$. Note that $\mathbf{MC}$ as well as $\mathbf{SC}$ may carry additional assets but for simplicity we will assume that staking and pegging is accomplished only via this single primary asset.

The presentation of the construction is organized as follows. First, in Section 6.4.1 we introduce a novel cryptographic primitive, *ad-hoc threshold multisignature (ATMS)*, which is the fundamental building block for cross-chain certification. Afterwards, in Section 6.4.3 we use it as a black box to build secure pegged ledgers with respect to concrete instantiations of the functions merge and effect and a validity language $\mathbb{V}_{\mathfrak{A}}$ for asset $\mathfrak{A}$ given in Section 6.4.2. Finally, we discuss specific instantiations of ATMS in Section 6.5.

6.4.1 Ad-Hoc Threshold Multisignatures

We introduce a new primitive, *ad-hoc threshold multisignatures (ATMS)*, which borrow properties from multisignatures and threshold signatures and are ad-hoc in the sense that signers need to be selected on the fly from an existing key set. In Section 6.4.3 we describe how ATMS are useful for periodically updating the "anchor of trust" that the mainchain parties have w.r.t. the sidechain they are not following.

ATMS are parametrized by a threshold t. On top of the usual digital signatures functionality, ATMS also provide a way to: (1) aggregate the public keys of a subset of these parties into a single aggregate public key avk; (2) check that a given avk was created using the right sequence of individual public keys; and (3) aggregate $t' \geq t$ individual signatures from t' of the parties into a single aggregate signature that can then be verified using avk, which is impossible if less than t individual signatures are used.

The definition of an ATMS is given below.

Definition 17. A *t-ATMS* is a tuple of algorithms $\Pi = (\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathsf{AKey}, \mathsf{ACheck}, \mathsf{ASig}, \mathsf{AVer})$ where:

$\mathsf{PGen}(1^\kappa)$ is the parameter generation algorithm that takes the security parameter 1^κ and returns system parameters $\mathcal{P}$.

$\mathsf{Gen}(\mathcal{P})$ is the key-generation algorithm that takes $\mathcal{P}$ and produces a public/private key pair (vk_i, sk_i) for the party invoking it.

$\mathsf{Sig}(sk_i, m)$ is the signature algorithm as in an ordinary signature scheme: it takes a private key and a message and produces a (so-called *local*) signature σ.

$\mathsf{Ver}(m, pk_i, \sigma)$ is the verification algorithm that takes a public key, a message and a signature and returns *true* or *false*.

$\mathsf{AKey}(\mathcal{VK})$ is the key aggregation algorithm that takes a sequence of public keys $\mathcal{VK}$ and aggregates them into an *aggregate public key* avk.

$\mathsf{ACheck}(\mathcal{VK}, avk)$ is the aggregation-checking algorithm that takes a public key sequence $\mathcal{VK}$ and an aggregate public key avk and returns *true* or *false*, determining whether $\mathcal{VK}$ were used to produce avk.

$\mathsf{ASig}\,(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle)$ is the signature-aggregation algorithm that takes a message m, a sequence of public keys $\mathcal{VK}$ and a sequence of d pairs $\langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle$ where each σ_i is a local signature on m verifiable by vk_i and each vk_i is in a distinct position within $\mathcal{VK}$, ASig combines these into a multisignature σ that can later be verified with respect to the aggregate public key avk produced from $\mathcal{VK}$ (as long as $d \geq t$, see below).

$\mathsf{AVer}(m, avk, \sigma)$ is the aggregate-signature verification algorithm that takes a message m, an aggregate public key avk, and a multisignature σ, and returns *true* or *false*.

Definition 18 (ATMS correctness)**.** Let Π be a t-ATMS scheme initialized with $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$, let (vk_1, sk_1), $\cdots, (vk_n, sk_n)$ be a sequence of keys generated via $\mathsf{Gen}(\mathcal{P})$, let $\mathcal{VK}$ be a sequence containing (not necessarily unique) keys from the above and avk be generated by invoking $avk \leftarrow \mathsf{AKey}(\mathcal{VK})$. Let m be any message and let $\langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle$ be any sequence of key/signature pairs provided that $d \geq t$ and every vk_i appears in a unique position in the sequence $\mathcal{VK}$, where σ_i is generated as $\sigma_i = \mathsf{Sig}(sk_i, m)$. Let $\sigma \leftarrow \mathsf{ASig}\,(m, \mathcal{VK}, \langle (vk_1, \sigma_1), \cdots, (vk_d, \sigma_d) \rangle)$. The scheme Π is *correct* if for every such message and sequence the following hold:

1. $\mathsf{Ver}(m, vk_i, \sigma_i)$ is *true* for all i;
2. $\mathsf{ACheck}(\mathcal{VK}, avk)$ is *true*;
3. $\mathsf{AVer}(m, avk, \sigma)$ is *true*.

We define the security of an ATMS in the definition below, via a cryptographic game given in Algorithm 1.

Definition 19 (Security)**.** A *t-ATMS* scheme $\Pi = (\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathsf{AKey}, \mathsf{ACheck}, \mathsf{ASig}, \mathsf{AVer})$ is *secure* if for any PPT adversary $\mathcal{A}$ and any polynomial p there exists some negligible function negl such that $\Pr[\mathsf{ATMS}_{\Pi,\mathcal{A}}(\kappa, p(\kappa)) = 1] < \mathsf{negl}(\kappa)$.

The quantity q in the ATMS game counts how many keys the adversary is in control of among her chosen keys keys which will be used for aggregate-signature verification. The sequence keys can contain both adversarially-generated keys as well as some of the keys $\mathcal{VK}$ honestly generated by the challenger. The variable q counts the number of adversarially controlled keys in keys. This includes those keys in keys for which the adversary has obtained a signature for the message in question (through the use of the oracle $\mathcal{O}^{\mathsf{sig}}(\cdot)$) or which the adversary has corrupted completely (through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$), as well as those keys which have been generated by the adversary herself and therefore are not in $\mathcal{VK}$.

It is straightforward to see that if Π is a secure ATMS, then the tuple $(\mathsf{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ is a EUF-CMA-secure signature scheme.

Looking ahead, note that since the AKey algorithm is only invoked with the public keys of the participants, it can be invoked by anyone, not just the parties who hold the respective secret keys, as long as the public portion of their keys is published. Furthermore, notice that the above games allow the adversary to generate more public/private key pairs of their own and combine them at will.

Having defined the ATMS primitive, we will now describe a sidechain construction that uses it. Concrete instantiations of the ATMS primitive are presented in Section 6.5.

Algorithm 1 The ATMS game

1: **function** $\mathrm{ATMS}(\kappa, p)$
2: $\mathcal{VK} \leftarrow \epsilon; \mathcal{SK} \leftarrow \epsilon; Q^{\mathsf{sig}} \leftarrow \emptyset; Q^{\mathsf{cor}} \leftarrow \emptyset$
3: $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$
4: $(m, \sigma, avk, \mathsf{keys}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{gen}}, \mathcal{O}^{\mathsf{sig}}(\cdot,\cdot), \mathcal{O}^{\mathsf{cor}}(\cdot)}(\mathcal{P})$
5: $q \leftarrow 0$
6: **for** vk in keys **do**
7: **if** $vk \notin \mathcal{VK} \vee vk \in Q^{\mathsf{sig}}[m] \cup Q^{\mathsf{cor}}$ **then**
8: $q \leftarrow q + 1$
9: **end if**
10: **end for**
11: **return** $\mathsf{AVer}(m, avk, \sigma) \wedge \mathsf{ACheck}(\mathsf{keys}, avk) \wedge q < t$
12: **end function**
13: **function** $\mathcal{O}^{\mathsf{gen}}$
14: $(vk, sk) \leftarrow \mathsf{Gen}(\mathcal{P})$
15: $\mathcal{VK} \leftarrow \mathcal{VK} \,\|\, vk$
16: $\mathcal{SK} \leftarrow \mathcal{SK} \,\|\, sk$
17: **return** vk
18: **end function**
19: **function** $\mathcal{O}^{\mathsf{sig}}(i, m)$
20: $Q^{\mathsf{sig}}[m] \leftarrow Q^{\mathsf{sig}}[m] \cup \{\mathcal{VK}[i]\}$
21: **return** $\mathsf{Sig}(\mathcal{SK}[i], m)$
22: **end function**
23: **function** $\mathcal{O}^{\mathsf{cor}}(i)$
24: $Q^{\mathsf{cor}} \leftarrow Q^{\mathsf{cor}} \cup \{\mathcal{VK}[i]\}$
25: **return** $\mathcal{SK}[i]$
26: **end function**

6.4.2 A Concrete Asset $\mathfrak{A}$

We now present an example of a simple fungible asset with fixed supply, which we denote $\mathfrak{A}$, and describe its validity language $\mathbb{V}_{\mathfrak{A}}$. This will be the asset (and validity language) considered in our construction and proof. While $\mathbb{V}_{\mathfrak{A}}$ is simple and natural, it allows us to exhibit the main features of our security treatment and illustrate how it can be applied to more complex languages such as those capable of capturing smart contracts; we omit such extensions in this version. Note that our language is account-based, but a UTXO-based validity language can be considered in a similar manner.

Instantiating $\mathbb{V}_{\mathsf{A}}$.

The validity language $\mathbb{V}_{\mathfrak{A}}$ for the asset $\mathfrak{A}$ considers two ledgers: the mainchain ledger $\mathbf{L}_0 \triangleq \mathbf{MC}$ and the sidechain ledger $\mathbf{L}_1 \triangleq \mathbf{SC}$. For this asset, every transaction $\mathsf{tx} \in \mathcal{T}_{\mathfrak{A}}$ has the form $\mathsf{tx} = (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma)$, where:

- txid is a transaction identifier that prevents replay attacks. We assume that txid contains sufficient information to identify lid by inspection and that this is part of syntactic transaction validation.
- $\mathsf{lid} \in \{0, 1\}$ is the ledger index where the transaction belongs.
- $\mathsf{send} \in \{0, 1\}$ is the index of the sender ledger $\mathbf{L}_{\mathsf{send}}$ and sAcc is an account on this ledger, this is the sender account. For simplicity, we assume that sAcc is the public key of the account.
- $\mathsf{rec} \in \{0, 1\}$ is the index of the recipient ledger $\mathbf{L}_{\mathsf{rec}}$ and rAcc is an account (again represented by a public key) on this ledger, this is the recipient account. We allow either $\mathbf{L}_{\mathsf{send}} = \mathbf{L}_{\mathsf{rec}}$, which denotes a *local transaction*, or $\mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$, which denotes a *remote transaction* (i.e., a cross-ledger transfer).
- v is the amount to be transferred.
- σ is the signature of the sender, i.e. made with the private key corresponding to the public key sAcc on the plaintext $(\mathsf{txid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v)$.

The correctness of lid is enforced by the ledgers, i.e., for both $i \in \{0, 1\}$ the set $\mathcal{T}_{\mathfrak{A}, \mathbf{L}_i}$ only contains transactions with $\mathsf{lid} = i$. Note that although we sometimes notationally distinguish between an account and the public key that is associated with it, for simplicity we will assume that these are either identical or can always be derived from one another (this assumption is not essential for our construction).

The membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$ is presented in Algorithm 2. It processes the sequence of transactions $(\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_m)$ given to it as input in their order. Assuming transactions are syntactically valid, the function verifies for each transaction tx_i the freshness of txid, validity of the signature, and availability of sufficient funds on the sending account. For an intra-ledger transaction (i.e., one that has $\mathsf{send} = \mathsf{rec}$), these are all the performed checks.

More interestingly, $\mathbb{V}_{\mathfrak{A}}$ also allows for cross-ledger transfers. Such transfers are expressed by a pair of transactions in which $\mathsf{send} \neq \mathsf{rec}$. The first transaction appears in $\mathsf{lid} = \mathsf{send}$, while the second transaction appears in $\mathsf{lid} = \mathsf{rec}$. The two transactions are identical except for this change in lid (this is the only exception to the txid-freshness requirement). Every receiving transaction has to be preceded by a matching sending transaction. Cross-chain transactions have to, similarly to intra-ledger transactions, conform to laws of balance conservation.

Note that $\mathbb{V}_{\mathfrak{A}}$ does not require that every "sending" cross-ledger transaction on the sender ledger is matched by a "receiving" transaction on the receiving ledger. Hence, if the asset $\mathfrak{A}$ is sent from ledger $\mathbf{L}_{\mathsf{send}}$ but has not yet arrived on $\mathbf{L}_{\mathsf{rec}}$ then validity for this asset is *not* violated. All the validity language ensures is that appending the $\mathsf{sidechain_receive}$ transaction to the rec will eventually be a valid way to extend the receiving ledger, as long as the $\mathsf{sidechain_send}$ transaction has been included in send.

Algorithm 2 The transaction sequence validator (membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$).

1: **function** valid-seq($\vec{\mathsf{tx}}$)
2: BALANCE $\leftarrow$ Initial stake distribution; seen $\leftarrow \emptyset$
3: ▷ *Traverse transactions in order*
4: **for** $\mathsf{tx} \in \vec{\mathsf{tx}}$ **do**
5: ▷ *Destructure* tx *into its constituents*
6: $(\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma) \leftarrow \mathsf{tx}$
7: **if** $\neg\text{valid}(\sigma)$ **then**
8: **return** false
9: **end if**
10: **if** $\mathsf{lid} = \mathsf{send}$ **then**
11: ▷ *Replay protection*
12: **if** $\mathsf{seen}[\mathsf{txid}] \neq 0$ **then**
13: **return** false
14: **end if**
15: ▷ *Law of conservation*
16: **if** $\text{BALANCE}[\mathsf{send}][\mathsf{sAcc}] - v < 0$ **then**
17: **return** false
18: **end if**
19: **else**
20: ▷ *The case* $\mathsf{lid} = \mathsf{rec} \neq \mathsf{send}$
21: **if** $\mathsf{seen}[\mathsf{txid}] \neq 1$ **then**
22: **return** false
23: **end if**
24: ▷ *Cross-ledger validity*
25: $\mathsf{tx}' \leftarrow \mathsf{effect}^{-1}_{\mathbf{L}_{(1-\mathsf{lid})} \to \mathbf{L}_{\mathsf{lid}}}(\mathsf{tx})$
26: **if** tx' has *not* appeared before **then**
27: **return** false
28: **end if**
29: **end if**
30: **if** $\mathsf{seen}[\mathsf{txid}] = 0$ **then**
31: ▷ *Update sender balance when money departs*
32: $\text{BALANCE}[\mathsf{send}][\mathsf{sAcc}] -= v$
33: **end if**
34: ▷ *Update receiver balance when money arrives*
35: **if** $(\mathsf{seen}[\mathsf{txid}] = 0 \wedge \mathsf{send} = \mathsf{rec}) \vee$
$(\mathsf{seen}[\mathsf{txid}] = 1 \wedge \mathsf{send} \neq \mathsf{rec})$ **then**
36: $\text{BALANCE}[\mathsf{rec}][\mathsf{rAcc}] += v$
37: **end if**
38: $\mathsf{seen}[\mathsf{txid}] += 1$
39: **end for**
40: **return** true
41: **end function**

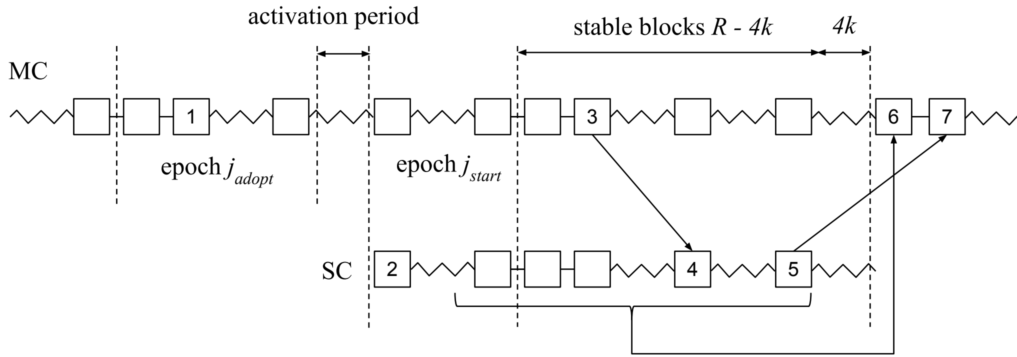


Figure 6.2: Our sidechain construction. Blocks are shown as rectangles. Adjacent blocks connect with straight lines. Squiggly lines indicate some blocks are omitted. $\mathbf{MC}$ is at the top, $\mathbf{SC}$ at the bottom. Epochs are separated by dashed lines. $e_{j_{\text{adopt}}}$ is the epoch of first signalling; $e_{j_{\text{start}}}$ is the activation epoch. Blocks of interest: 1. The first block signalling $\mathbf{SC}$ awareness; 2. The $\mathbf{SC}$ genesis block; 3. A $\mathsf{tx}_{\mathsf{send}}$ transaction for a deposit; 4. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for a deposit; 5. A $\mathsf{tx}_{\mathsf{send}}$ transaction for withdrawal; 6. A sc_cert transaction signalling trust transition within $\mathbf{SC}$ and certifying pending withdrawals; 7. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for withdrawal, certified in a sc_cert transaction e.g. in block 6.

Instantiating $\mathsf{effect}_{\mathbf{L}_i \to \mathbf{L}_j}$**.**

For the simple asset $\mathfrak{A}$ outlined above, every cross-ledger transfer is a "sending" transaction tx with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$ appearing in $\mathbf{L}_{\mathsf{send}}$, and its effect transaction is a "receiving" transaction tx' with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{rec}} \neq \mathbf{L}_{\mathsf{send}}$ in $\mathbf{L}_{\mathsf{rec}}$ that is otherwise identical (except for the different $\mathsf{lid}' = 1 - \mathsf{lid}$). Hence, we define $\mathsf{effect}_{\mathbf{L}_{\mathsf{send}} \to \mathbf{L}_{\mathsf{rec}}}(\mathsf{tx}) = \mathsf{tx}'$ exactly for all these transactions and no other.

Instantiating $\mathsf{merge}(\cdot)$**.**

It is easy to construct a canonical function $\mathsf{merge}(\cdot)$ once we see its inputs not only as ledger states (i.e., sequences of transactions) but we also exploit the additional structure of the blockchains carrying those ledgers. The *canonical merge* of the set of ledger states $\mathcal{L}$ is the lexicographically minimum topologically sound merge, in which transactions of ledger L_i are compared favourably to transactions in L_j if $i < j$. However, note that the construction we provide below will work for any topologically sound merge function.

One can easily observe the following statement.

Proposition 8. *The validity language $\mathbb{V}_{\mathfrak{A}}$ is correct (according to Definition 15) with respect to the* merge *function defined above.*

6.4.3 The Sidechain Construction

We now describe the procedures for running a sidechain in the configuration outlined at the beginning of this section: with independent staking on $\mathbf{MC}$ and merged staking on $\mathbf{SC}$; direct observation of $\mathbf{MC}$ and cross-chain certification of $\mathbf{SC}$. We describe the sidechain's creation, maintenance, and the way assets can be transferred to it and back. The protocol we describe below is quite complex, we hence choose to describe different parts of the protocol in differing levels of detail. This level is always chosen with the intention to allow the reader to easily fill in the details. A graphical depiction of our construction that can serve as a reference is given in Figure 6.2.

Notation.

Where applicable, we denote the analogues of the mainchain objects on the sidechain with an additional overline. In our pseudocode, we use the statement "**post** tx **to** $\mathbf{L}$" to refer to the action of broadcasting the transaction tx to the maintainers of the ledger $\mathbf{L}$ so that they include it in the ledger eventually as prescribed by the protocol. Unless indicated otherwise, we also denote by MC (resp. SC) the current *ledger state* of the ledger $\mathbf{MC}$ (resp. $\mathbf{SC}$) as viewed by the party executing the protocol. Similarly, we denote by $\mathsf{C}_{\mathbf{MC}}$ (resp. $\mathsf{C}_{\mathbf{SC}}$) the currently held *chain* corresponding to the ledger $\mathbf{MC}$ (resp. $\mathbf{SC}$). Hence, for example MC always represents the state stored in the stable part of the chain $\mathsf{C}_{\mathbf{MC}}$.

Helper Transactions and Data.

The construction uses a set of *helper transactions* which can be included in both blockchains, but do not get reported in the respective ledgers. These helper transactions store the appropriate metadata which is implementation-specific and allow the pegging functionality to be maintained. The transaction types sidechain_support, sidechain_certificate, sidechain_success and sidechain_failure, whose nature will be detailed later, are of this kind. Moreover, our concrete implementation of pegged ledgers extends certain transactions with additional information (such as Merkle-tree inclusion proofs) that are, for convenience, understood to be stripped off these transactions when the blockchain is interpreted as a ledger.

Initialization.

The creation of a new sidechain $\mathbf{SC}$ starts by any of the stakeholders of the mainchain adopting the code that implements the sidechain. This action does not require the stakeholders to put stake on the sidechain but merely to run the code to support it (e.g. by installing a pluggable module into their client software). In the following this is referred to as "adopting the sidechain" and captured by the predicate SidechainAdoption. The adoption is announced at the mainchain by a special transaction detailed below. Each sidechain is identified by a unique identifier $\mathsf{id}_{\mathbf{SC}}$.

Let j_{adopt} denote the epoch on $\mathbf{MC}$ when the first adoption transaction has appeared; the sidechain $\mathbf{SC}$ – if its activation succeeds as discussed below – will start at the beginning of some later epoch j_{start} and will have its slots and epochs synchronized with $\mathbf{MC}$. The software module implementing the sidechain comes with a set of deterministic rules describing the requirements for the successful activation of the sidechain, as well as for determining j_{start}. These rules are sidechain-specific and are captured in a predicate ActivationSuccess and a function ActivationEpoch, respectively. One typical such example is the following: the sidechain starts at the beginning of $\mathbf{MC}$-epoch j_{start} for the smallest j_{start} that satisfies: (i) $j_{\mathsf{start}} - j_{\mathsf{adopt}} > c_1$; (ii) at least c_2-fraction of stake on $\mathbf{MC}$ is controlled by stakeholders that have adopted $\mathbf{SC}$; for some constants c_1,c_2. Additionally, if such a successful activation does not occur until a failure condition captured by a predicate ActivationFailure is met (e.g. until a predetermined period of $c_3 > c_1$ epochs has passed), the sidechain initialization is aborted.

The activation process then follows the steps outlined below, the detailed description is given in Algorithm 3).

First, every stakeholder U_i of $\mathbf{MC}$ (holding a key pair (vk, sk)) that supports the sidechain posts a special transaction $[\textsc{sidechain_support}, \mathsf{id}_{\mathbf{SC}}, vk, vk']$, signed by sk into the mainchain. Here vk' is a public key from an ATMS key pair freshly generated by U_i; its role is explained in Section 6.4.3 below.

If the sidechain activation succeeds, then during the first slot of epoch j_{start} the stakeholders of $\mathbf{MC}$ that support $\mathbf{SC}$ construct the genesis block $\overline{\mathsf{G}} = (\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \bar{\eta}_{j_{\mathsf{start}}} \triangleq H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}}), \mathcal{P}, avk^{j_{\mathsf{start}}})$ for $\mathbf{SC}$ where H is hash function. $\eta_{j_{\mathsf{start}}}$ is the randomness for leader election on $\mathbf{MC}$ in epoch j_{start} (derived on $\mathbf{MC}$ in epoch $j_{\mathsf{start}} - 1$). It is reused to compute the initial sidechain randomness $\bar{\eta}_{j_{\mathsf{start}}}$ as well, further $\bar{\eta}_{j'}$ for $j' > j_{\mathsf{start}}$ are determined independently on $\mathbf{SC}$ using the Ouroboros coin-tossing protocol.[6] Furthermore, $\mathcal{P}$ and $avk^{j_{\mathsf{start}}}$ are public parameters and an aggregated public key of an ATMS scheme; their creation and role is discussed in

[6]This can be interpreted as using $\mathbf{MC}$ to implement the setup functionality needed to bootstrap $\mathbf{SC}$.

Algorithm 3 Sidechain Initialization procedures.

The algorithm is run by every stakeholder U that adopted the sidechain. We denote by (vk, sk) its public and private keys.

1: **upon** $\mathsf{SidechainAdoption}(\mathsf{id}_{\mathbf{SC}})$ **do**
2: $\quad$ $\mathsf{sidechain_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initializing}$
3: $\quad$ $(vk', sk') \leftarrow \mathsf{Gen}(\mathcal{P})$
4: $\quad$ $\sigma \leftarrow \mathsf{Sig}_{sk}[\text{SIDECHAIN_SUPPORT}, \mathsf{id}_{\mathbf{SC}}, vk, vk']$
5: $\quad$ **post** $[\text{SIDECHAIN_SUPPORT}, \mathsf{id}_{\mathbf{SC}}, vk, vk', \sigma]$ **to** $\mathbf{MC}$
6: **end upon**
7: **upon** $\mathbf{MC}.\mathsf{NewEpoch}()$ **do**
8: $\quad$ $j \leftarrow \mathbf{MC}.\mathsf{EpochIndex}()$
9: $\quad$ **if** $\mathsf{sidechain_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initializing}$ **then**
10: $\quad\quad$ **if** $\mathsf{ActivationFailure}()$ **then**
11: $\quad\quad\quad$ $\mathsf{sidechain_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{failed}$
12: $\quad\quad\quad$ **post** $\mathsf{sidechain_failure}(\mathsf{id}_{\mathbf{SC}})$ **to** $\mathbf{MC}$
13: $\quad\quad$ **else if** $\mathsf{ActivationSuccess}()$ **then**
14: $\quad\quad\quad$ $\mathsf{sidechain_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initialized}$
15: $\quad\quad\quad$ $j_{\mathsf{start}} \leftarrow \mathsf{ActivationEpoch}()$
16: $\quad\quad\quad$ Post $\mathsf{sidechain_success}(\mathsf{id}_{\mathbf{SC}})$ to $\mathbf{MC}$
17: $\quad\quad$ **end if**
18: $\quad$ **end if**
19: $\quad$ **if** $\mathsf{sidechain_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initialized} \wedge j = j_{\mathsf{start}}$ **then**
20: $\quad\quad$ $\bar{\eta}_{j_{\mathsf{start}}} \leftarrow H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}})$
21: $\quad\quad$ $\mathcal{VK}_{j_{\mathsf{start}}} \leftarrow 2k$ last slot leaders of $e_{j_{\mathsf{start}}}$ in $\mathbf{SC}$
22: $\quad\quad$ $avk^{j_{\mathsf{start}}} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j_{\mathsf{start}}})$
23: $\quad\quad$ $\overline{\mathsf{G}} \leftarrow \left(\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \bar{\eta}_{j_{\mathsf{start}}}, \mathcal{P}, avk^{j_{\mathsf{start}}}\right)$
24: $\quad\quad$ $\mathsf{C}_{\mathbf{SC}} \leftarrow (\overline{\mathsf{G}})$
25: $\quad$ **end if**
26: **end upon**

Section 6.4.3 below. Note that $\overline{\mathsf{G}}$ is defined mostly for notational compatibility, as $\overline{\mathsf{SD}}_{j_{\mathsf{start}}}$ is empty at this point anyway. $\overline{\mathsf{G}}$ can be constructed as soon as $\eta_{j_{\mathsf{start}}}$ is known and stable.

The stakeholders that adopted $\mathbf{SC}$ post into $\mathbf{MC}$ a transaction $\mathsf{sidechain_success}(\mathsf{id}_{\mathbf{SC}})$ to signify that $\mathbf{SC}$ has been initialized. If the sidechain creation expires, then, after the first block of the next epoch after expiration occurs, the stakeholders of $\mathbf{MC}$ that supported $\mathbf{SC}$ post the transaction $\mathsf{sidechain_failure}(\mathsf{id}_{\mathbf{SC}})$ to $\mathbf{MC}$. We assume that both predicates ActivationSuccess and ActivationFailure can be evaluated based on the state of $\mathbf{MC}$ only, and hence spurious success/failure transactions will be considered invalid.

Maintenance.

Once the sidechain is created, both the mainchain and the sidechain need to be maintained by their respective set of stakeholders (detailed below) running their respective instance of the Ouroboros protocol.

In the case of the mainchain, the maintenance procedure is given in Algorithm 4. This algorithm is run by all stakeholders controlling stake that is recorded on the mainchain. Each stakeholder, on every new slot, collects all the candidate $\mathbf{MC}$-chains from the network (modelled via the Diffuse functionality) and filters them for both consensus-level validity (using $\mathbf{MC}$.ValidateConsensusLevel) and transaction validity (using the $\textsc{verifier}_{\mathbf{MC}}$ predicate given in Algorithm 5). Out of the remaining valid chains, he chooses his new state $\mathsf{C}_{\mathbf{MC}}$ via PickWinningChain. Then the stakeholder evaluates whether he is an eligible leader for this slot, basing its selection on the stake distribution SD_j and randomness η_j, which are determined once per epoch in accordance with the Ouroboros protocol. If the stakeholder finds out he is a slot leader, he creates a new block B by including all transactions currently valid with respect to $\mathsf{C}_{\mathbf{MC}}$ (as per the predicate $\textsc{verifytx}_{\mathbf{MC}}$ given also in Algorithm 5), appends it to the chain $\mathsf{C}_{\mathbf{MC}}$ and diffuses the result[7] for other parties to adopt.

The maintenance procedure for $\mathbf{SC}$ is similar, hence we only describe here how it differs from Algorithm 4. Most importantly, it is executed by all stakeholders who have adopted $\mathbf{SC}$, irrespectively of whether they own any stake on $\mathbf{SC}$. Recall that the slots and epochs of the $\mathbf{SC}$-instance of Ouroboros are aligned with the slots and epochs of $\mathbf{MC}$.

The first difference is that all ocurrences of $\mathbf{MC}$ and $\mathsf{C}_{\mathbf{MC}}$ are naturally replaced by $\mathbf{SC}$ and $\mathsf{C}_{\mathbf{SC}}$, respectively. This also means that the validity of received chains (resp. transactions), determined on line 13 (resp. 21), is decided based on predicate $\textsc{verifier}_{\mathbf{SC}}(\cdot, \mathsf{C}_{\mathbf{MC}})$ (resp. $\textsc{verifytx}_{\mathbf{SC}}(\cdot)$) instead of the predicate $\textsc{verifier}_{\mathbf{MC}}(\cdot)$ (resp. $\textsc{verifytx}_{\mathbf{MC}}(\cdot)$). Additionally, note that $\textsc{verifytx}_{\mathbf{SC}}$ must be called with a sequence of transactions containing both the transactions in $\mathbf{SC}$ as well as the transactions in $\mathbf{MC}$ interspersed and timestamped, similarly to the way done in Line 2 of Algorithm 7. This is straightforward to implement, as the sidechain maintainers also directly observe the mainchain. The predicates $\textsc{verifytx}_{\mathbf{SC}}$ and $\textsc{verifier}_{\mathbf{SC}}$ are given in Algorithms 6 and 7, respectively.

Second, instead of the stake distribution SD_j determined on line 6, a different distribution $\overline{\mathsf{SD}}^*_j$ is determined to be used for slot leader selection in the j-th epoch of the sidechain. The distribution $\overline{\mathsf{SD}}^*$ contains all stake belonging to stakeholders that have adopted $\mathbf{SC}$, irrespectively of whether this stake is located on $\mathbf{MC}$ or $\mathbf{SC}$ (we call such stake $\mathbf{SC}$-*aware*). It can be obtained by combining the distribution $\overline{\mathsf{SD}}$ as recorded in $\mathbf{SC}$ with the distribution of $\mathbf{SC}$-aware stake on $\mathbf{MC}$ (which is known to $\mathbf{SC}$-maintainers via direct observation of $\mathbf{MC}$). Note that the distribution used for epoch j reflects the stake distribution of $\mathbf{SC}$-aware stake in the past, namely by slot $4k$ of epoch $j-1$, just as in $\mathbf{MC}$. Naturally, this also implies that the fourth parameter for the SlotLeader predicate on line 17 is $\overline{\mathsf{SD}}^*_j$ instead of SD_j.

Finally, the block construction procedure on line 23 is adjusted so that in the last $2k$ slots of each epoch, the created blocks on the sidechain also contain an additional ATMS signature of a so-called sidechain certificate (how this certificate is constructed and used will be described below). Hence, whenever $sl \mod R > 10k$,

[7] As in [KRDO17, DGKR18], we simplify our presentation by diffusing the complete chains, although a practical implementation would only diffuse the block B.

Algorithm 4 Mainchain maintenance procedures.

The algorithm is run by every stakeholder U with stake on $\mathbf{MC}$ in every epoch $j \geq j_{\mathsf{start}}$, sk denotes the secret key of U. An analogous mainchain-maintaining procedure was running also before j_{start} and is omitted.

1: **upon** $\mathbf{MC}.\mathsf{NewSlot}()$ **do**
2: $sl \leftarrow \mathbf{MC}.\mathsf{SlotIndex}()$
3: ▷ *First slot of a new epoch*
4: **if** $sl \mod R = 1$ **then**
5: $j \leftarrow \mathbf{MC}.\mathsf{EpochIndex}()$
6: $\mathsf{SD}_j \leftarrow \mathbf{MC}.\mathsf{GetDistr}(j)$
7: $\eta_j \leftarrow \mathbf{MC}.\mathsf{GetRandomness}(j)$
8: **end if**
9: $\mathcal{C} \leftarrow$ chains received via $\mathsf{Diffuse}$
10: ▷ *Consensus-level validation*
11: $\mathcal{C}_{\mathsf{valid}} \leftarrow \mathsf{Filter}(\mathcal{C}, \mathbf{MC}.\mathsf{ValidateConsensusLevel})$
12: ▷ *Transaction-level validation*
13: $\mathcal{C}_{\mathsf{validtx}} \leftarrow \mathsf{Filter}(\mathcal{C}_{\mathsf{valid}}, \textsc{verifier}_{\mathbf{MC}}(\cdot))$
14: ▷ *Apply chain selection rule*
15: $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathbf{MC}.\mathsf{PickWinningChain}(\mathsf{C}_{\mathbf{MC}}, \mathcal{C}_{\mathsf{validtx}})$
16: ▷ *Decide slot leadership based on* SD_j *and* η_j
17: **if** $\mathbf{MC}.\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ **then**
18: $\mathsf{prev} \leftarrow H(\mathsf{C}_{\mathbf{MC}}[-1])$
19: $\vec{\mathsf{tx}}_{\mathsf{state}} \leftarrow$ transaction sequence in $\mathsf{C}_{\mathbf{MC}}$
20: $\vec{\mathsf{tx}} \leftarrow$ current transactions in mempool
21: $\vec{\mathsf{tx}}_{\mathsf{valid}} \leftarrow \textsc{verifytx}_{\mathbf{MC}}(\vec{\mathsf{tx}}_{\mathsf{state}} \parallel \vec{\mathsf{tx}})[|\vec{\mathsf{tx}}_{\mathsf{state}}| :]$
22: $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}})$
23: $B \leftarrow (\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}}, \sigma)$
24: $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathsf{C}_{\mathbf{MC}} \parallel B$
25: $\mathsf{Diffuse}(\mathsf{C}_{\mathbf{MC}})$
26: **end if**
27: **end upon**

Algorithm 5 The $\mathbf{MC}$ verifier.

```
 1: function VERIFYTX_MC(tx⃗)
 2:     bal ← initial stake; avk ← initial aggregate key
 3:     seen ← ∅; pool ← 0; pfs_mtrs ← ∅; pfs_used ← ∅
 4:     for tx ∈ tx⃗ do
 5:         if type(tx) = sc_cert then
 6:             (m, σ) ← tx
 7:             if ¬AVer(m, avk, σ) then
 8:                 continue
 9:             end if
10:             (txs_root, avk′) ← m
11:             avk ← avk′
12:             pfs_mtrs[txs_root] ← true
13:         else
14:             (txid, lid, (send, sAcc), (rec, rAcc), v, σ) ← tx
15:             m ← (txid, lid, (send, sAcc), (rec, rAcc), v)
16:             if ¬Ver(m, sAcc, σ) ∨ seen[txid] ≠ 0 then
17:                 continue
18:             end if
19:             if lid = send then
20:                 if bal[sAcc] − v < 0 then
21:                     continue
22:                 end if
23:                 bal[sAcc] −= v
24:             else if send ≠ rec then
25:                 π ← tx.π
26:                 (mtr, inclusion_pf) ← π
27:                 if π ∈ pfs_used ∨ mtr ∉ pfs_mtrs∨ ¬MTR-VER(mtr, inclusion_pf) then
28:                     continue
29:                 end if
30:             end if
31:             if lid = rec then
32:                 bal[rAcc] += v
33:             end if
34:             if send ≠ rec then
35:                 if lid = send then
36:                     pool −= v
37:                 else
38:                     pool += v
39:                 end if
40:             end if
41:         end if
42:         seen ← seen ∥ tx
43:     end for
44:     return seen
45: end function
46: function VERIFIER_MC(C_mc)
47:     tx⃗ ← ∅
48:     for B ∈ C_mc do
49:         for tx ∈ B do
50:             tx⃗ ← tx⃗ ∥ tx
51:         end for
52:     end for
53:     return tx⃗ ≠ VERIFYTX_MC(tx⃗)
54: end function
```

line 23 is replaced by $B \leftarrow (\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}}, \sigma, \sigma_{\mathsf{sc_cert}_{j+1}})$ where $\sigma_{\mathsf{sc_cert}_{j+1}} = \mathsf{Sig}_{sk}(\mathsf{sc_cert}_{j+1})$, j is the current epoch index and $\mathsf{sc_cert}_{j+1}$ is the *sidechain certificate* that we formally describe in Algorithm 12). The role of

Algorithm 9 Constructing sidechain certificate $\mathsf{sc_cert}$.

The algorithm is run by every $\mathbf{SC}$-maintainer at the end of each epoch, j denotes the index of the ending epoch.

1: **function** $\mathsf{ConstructSCCert}(j)$
2: $T \leftarrow$ last $4k$ slots of e_{j-1} and first $R - 4k$ slots of e_j
3: $\vec{\mathsf{tx}} \leftarrow$ transactions included in $\mathbf{SC}$ during T
4: $\mathsf{pending}_{j+1} \leftarrow \left\{\mathsf{tx} \in \vec{\mathsf{tx}} \;:\; \mathsf{tx.send} \neq \mathsf{tx.rec}\right\}$
5: $\mathcal{VK}_{j+1} \leftarrow$ keys of last $2k$ $\mathbf{SC}$ slot leaders in e_{j+1}
6: $avk^{j+1} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j+1})$
7: $m \leftarrow \left(\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}\right)$
8: $\mathcal{VK}_j \leftarrow$ keys of last $2k$ $\mathbf{SC}$ slot leaders for e_j
9: $\sigma_{j+1} \leftarrow \mathsf{ASig}\left(m, \{(vk_i, \sigma_i)\}_{i=1}^{d}, \mathcal{VK}_j\right)$
10: $\mathsf{sc_cert}_{j+1} \leftarrow (\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}, \sigma_{j+1})$
11: **return** $\mathsf{sc_cert}_{j+1}$
12: **end function**

the certificate produced by the end of epoch $j-1$ to be included in $\mathbf{MC}$ at the beginning of epoch j (denoted $\mathsf{sc_cert}_j$) is to attest all the withdrawals that had their sending transactions included into $\mathbf{SC}$ in either the last $4k$ slots of e_{j-2} or the first $R - 4k$ slots of e_{j-1}. More details are given later.

Depositing to SC.

Once $\mathbf{SC}$ is initialized, cross-chain transfers to it can be made from $\mathbf{MC}$. A cross-chain transfer operation in this case consists of two transactions $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$ that both have $\mathsf{send} = \mathbf{MC}$, $\mathsf{rec} = \mathbf{SC}$, and all other fields are also identical, except that each tx_i for $i \in \{\mathsf{send}, \mathsf{rec}\}$ contains $\mathsf{lid} = i$. The *sending transaction* $\mathsf{tx}_{\mathsf{send}}$ is meant to be included in $\mathbf{MC}$, while the *receiving transaction* $\mathsf{tx}_{\mathsf{rec}}$ is meant to be included in $\mathbf{SC}$.

Whenever a stakeholder on $\mathbf{MC}$ that has adopted $\mathbf{SC}$ wants to transfer funds to $\mathbf{SC}$, she diffuses $\mathsf{tx}_{\mathsf{send}}$ with the correct receiving account on $\mathbf{SC}$ and the desired amount. Honest slot leaders in $\mathbf{MC}$ include these transactions into their blocks just like any intra-chain transfer transactions. Maintainers of $\mathbf{MC}$ keep account of a variable $\mathsf{pool}_{\mathbf{SC}}$, initially set to zero. Whenever a $\mathsf{tx}_{\mathsf{send}}$ is included into $\mathbf{MC}$, they increase $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction.

When $\mathsf{tx}_{\mathsf{send}}$ becomes stable in $\mathbf{MC}$ (i.e., appears in MC, this happens at most $2k$ slots after its inclusion), the stakeholder creates and diffuses the corresponding $\mathsf{tx}_{\mathsf{rec}}$ which credits the respective amount of coins to rAcc in $\mathbf{SC}$, to be included into $\mathbf{SC}$. In practice, this is akin to a coinbase transaction, as the money was not transferred from an existing $\mathbf{SC}$ account.

Note that depositing from $\mathbf{MC}$ to $\mathbf{SC}$ is relatively fast; it merely requires a reliable inclusion of $\mathsf{tx}_{\mathsf{send}}$ into $\mathbf{MC}$ and consequently of $\mathsf{tx}_{\mathsf{rec}}$ into $\mathbf{SC}$, as guaranteed by the liveness of the underlying Ouroboros instances. The depositing algorithm code is shown in Algorithm 10.

Withdrawing to MC.

The withdrawal operation is more cumbersome than the depositing operation since not all nodes of $\mathbf{MC}$ have adopted (i.e., are aware of and follow) the sidechain $\mathbf{SC}$. As transactions, the withdrawals have the same structure as deposits, consisting of $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$, with the only difference that now they both have $\mathsf{send} = \mathbf{SC}$ and $\mathsf{rec} = \mathbf{MC}$. The sending transaction will be handled in the same way as in the case of deposits, but the receiving transaction requires a different certificate-based treatment, as detailed below.

Algorithm 10 Depositing from $\mathbf{MC}$ to $\mathbf{SC}$.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on $\mathbf{MC}$.

1: **function** $\mathsf{Send}(\mathsf{sAcc}, \mathsf{rAcc}, v)$
2: $\quad \mathsf{txid} \xleftarrow{\$} \{0,1\}^k$
3: $\quad \sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
4: $\quad \mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
5: $\quad$ **post** $\mathsf{tx}_{\mathsf{send}}$ **to** $\mathbf{MC}$
6: **end function**
7: **function** $\mathsf{Receive}(\mathsf{txid}, \mathsf{sAcc}, \mathsf{rAcc}, v)$
8: $\quad$ **wait until** $\mathsf{tx}_{\mathsf{send}} \in \mathsf{MC}$
9: $\quad \sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
10: $\quad \mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
11: $\quad$ **post** $\mathsf{tx}_{\mathsf{rec}}$ **to** $\mathbf{SC}$
12: **end function**

Whenever a stakeholder in $\mathbf{SC}$ wishes to withdraw coins from $\mathbf{SC}$ to $\mathbf{MC}$, she creates and diffuses the respective transaction $\mathsf{tx}_{\mathsf{send}}$ with the correct transfer details as before. If $\mathsf{tx}_{\mathsf{send}}$ is included in a block that belongs in one of the first $R-4k$ slots of some epoch then let j_{send} denote the index of this epoch, otherwise let j_{send} denote the index of the following epoch. The stakeholder then waits for the end of the epoch $e_{j_{\mathsf{send}}}$ to pass and $e_{j_{\mathsf{send}}+1}$ to begin.

At the beginning of $e_{j_{\mathsf{send}}+1}$, a special transaction called *sidechain certificate* $\mathsf{sc_cert}_{j_{\mathsf{send}}+1}$ is generated by the maintainers of $\mathbf{SC}$. It contains: (i) a Merkle-tree commitment to all withdrawal transactions $\mathsf{tx}_{\mathsf{send}}$ that were included into $\mathbf{SC}$ during last $4k$ slots of epoch $j_{\mathsf{send}}-1$ and the first $R-4k$ slots of epoch j_{send} (as these all are already stable by slot $R-2k$ of epoch j_{send}); (ii) other information allowing the maintainers of $\mathbf{MC}$ to inductively validate the certificate in every epoch. The construction of $\mathsf{sc_cert}$ is detailed below, for now assume that the transaction provides a proof that the included information about withdrawal transactions is correct. The transaction $\mathsf{sc_cert}$ is broadcast into the $\mathbf{MC}$ network to be included into $\mathbf{MC}$ at the beginning of $e_{j_{\mathsf{send}}+1}$ by the first honest slot leader.

The stakeholder who wishes to withdraw their money into $\mathbf{MC}$ now creates and diffuses the transaction $\mathsf{tx}_{\mathsf{rec}}$ to be included in $\mathbf{MC}$. This transaction is only included into $\mathbf{MC}$ if it is considered valid, which means: (1) it is properly signed; (2) it contains a Merkle inclusion proof confirming its presence in some already included sidechain certificate; (3) its amount is less or equal to the current value of $\mathsf{pool}_{\mathbf{SC}}$. If included, $\mathbf{MC}$-maintainers decrease the value of $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction. The code of the withdrawal algorithm is illustrated in Algorithm 11.

The certificate transaction.

We now describe the construction of the $\mathsf{sc_cert}$ transaction, also called the *sidechain certificate*, formally described in Algorithm 12). The role of the certificate produced by the end of epoch $j-1$ to be included in $\mathbf{MC}$ at the beginning of epoch j (denoted $\mathsf{sc_cert}_j$) is to attest all the withdrawals that had their sending transactions included into $\mathbf{SC}$ in either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_{j-1}. To maintain a chain of trust for the $\mathbf{MC}$ maintainers that cannot verify these transactions by observing $\mathbf{SC}$, we make use of ad-hoc threshold multisignatures introduced in Section 6.4.1. Namely, the $\mathsf{sc_cert}_j$ transaction also contains an aggregate key avk^j of an ATMS, and is signed by the previous aggregate key avk^{j-1} included in $\mathsf{sc_cert}_{j-1}$.

$\mathsf{sc_cert}_j$ is generated by $\mathbf{SC}$-maintainers and contains:

Algorithm 11 Withdrawing from $\mathbf{SC}$ to $\mathbf{MC}$.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on $\mathbf{SC}$.

1: **function** Send(sAcc, rAcc, v) ▷ Send v from sAcc on $\mathbf{SC}$ to rAcc on $\mathbf{MC}$
2: $\mathsf{txid} \xleftarrow{\$} \{0,1\}^k$
3: $\sigma \leftarrow \mathsf{Sig}_{sk}\,(\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v)$
4: $\mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \sigma)$
5: **post** $\mathsf{tx}_{\mathsf{send}}$ **to** $\mathbf{SC}$
6: **end function**
7: **function** Receive(txid, sAcc, rAcc, v)
8: **wait until** $\mathsf{tx}_{\mathsf{send}} \in \mathsf{C}_{\mathbf{SC}}$
9: $j' \leftarrow$ epoch where $\mathsf{C}_{\mathbf{SC}}$ contains $\mathsf{tx}_{\mathsf{send}}$
10: **if** ($\mathsf{tx}_{\mathsf{send}}$ included in slot $sl \leq R - 4$ of $e_{j'}$) **then**
11: $j_{\mathsf{send}} \leftarrow j'$
12: **else**
13: $j_{\mathsf{send}} \leftarrow j' + 1$
14: **end if**
15: **wait until** $\mathsf{sc_cert}_{j_{\mathsf{send}}+1} \in \mathsf{C}_{\mathbf{MC}}$
16: $\pi \leftarrow$ Merkle-tree proof of $\mathsf{tx}_{\mathsf{send}}$ in $\mathsf{sc_cert}_{j_{\mathsf{send}}+1}$
17: $\sigma \leftarrow \mathsf{Sig}_{sk}\,(\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi)$
18: $\mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi, \sigma)$
19: **post** $\mathsf{tx}_{\mathsf{rec}}$ **to** $\mathbf{MC}$
20: **end function**

Algorithm 12 Constructing sidechain certificate sc_cert.

The algorithm is run by every $\mathbf{SC}$-maintainer at the end of each epoch, j denotes the index of the ending epoch.

1: **function** ConstructSCCert(j)
2: $T \leftarrow$ last $4k$ slots of e_{j-1} and first $R - 4k$ slots of e_j
3: $\vec{\mathsf{tx}} \leftarrow$ transactions included in $\mathbf{SC}$ during T
4: $\mathsf{pending}_{j+1} \leftarrow \left\{\mathsf{tx} \in \vec{\mathsf{tx}} \,:\, \mathsf{tx.send} \neq \mathsf{tx.rec}\right\}$
5: $\mathcal{VK}_{j+1} \leftarrow$ keys of last $2k$ $\mathbf{SC}$ slot leaders in e_{j+1}
6: $avk^{j+1} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j+1})$
7: $m \leftarrow \left(\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}\right)$
8: $\mathcal{VK}_j \leftarrow$ keys of last $2k$ $\mathbf{SC}$ slot leaders for e_j
9: $\sigma_{j+1} \leftarrow \mathsf{ASig}\left(m, \{(vk_i, \sigma_i)\}_{i=1}^{d}, \mathcal{VK}_j\right)$
10: $\mathsf{sc_cert}_{j+1} \leftarrow (\left\langle \mathsf{pending}_{j+1} \right\rangle, avk^{j+1}, \sigma_{j+1})$
11: **return** $\mathsf{sc_cert}_{j+1}$
12: **end function**

- **The epoch index** j.
- **The pending transactions from SC to MC.** Let $\vec{\mathsf{tx}}$ be the sequence of all transactions which are included in $\mathbf{SC}$ during either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_j. All transactions in $\vec{\mathsf{tx}}$ that have $\mathbf{SC} = \mathsf{send} \neq \mathsf{rec} = \mathbf{MC}$ are picked up and combined into a list $\mathsf{pending}_j$ (sorted in the same order as in $\mathbf{SC}$). Let $\langle\mathsf{pending}_j\rangle$ denote a Merkle-tree commitment to this list.
- **The new ATMS key avk^j.** The key is created from the public keys of the slot leaders of the last $2k$ slots of the epoch j, using threshold $k+1$. Hence, it allows to verify whether a particular signature comes from $k+1$ out of these $2k$ keys.
- **Signature valid with respect to avk^{j-1}.**

The full $\mathsf{sc_cert}_j$ is therefore a tuple $\big(\langle\mathsf{pending}_j\rangle, avk^j, \sigma_j\big)$, where σ_j is an ATMS signature on the preceding elements that verifies using avk^{j-1}.

The certificate $\mathsf{sc_cert}_{j+1}$ is constructed as follows: Both the stake distribution $\overline{\mathsf{SD}}^*_{j+1}$ and the $\mathbf{SC}$-randomness $\bar{\eta}_{j+1}$ (and hence also the slot leader schedule for $\mathbf{SC}$ in epoch $j+1$) are determined by the states of the blockchains $\mathbf{MC}$ and $\mathbf{SC}$ by the end of slot $10k$ of epoch j. Therefore, during the last $2k$ slots of epoch j, the $2k$ elected slot leaders for these slots can already include a (local) signature on (their proposal of) $\mathsf{sc_cert}_{j+1}$ into the blocks they create. Given the deterministic construction of $\mathsf{sc_cert}_{j+1}$, all valid blocks ending up in the part of $\mathbf{SC}$-chain belonging to the last $2k$ slots of epoch j will contain a local signature on the same $\mathsf{sc_cert}_{j+1}$, and by the chain growth property of the underlying blockchain, there will be at least $k+1$ of them. Therefore, any party observing $\mathbf{SC}$ can now combine these signatures into an ATMS that can be later verified using the ATMS key avk^j, it can hence create the complete certificate $\mathsf{sc_cert}_{j+1}$ and serve it to the maintainers of $\mathbf{MC}$ for inclusion.

Transitioning trust.

As already outlined above, our construction uses ATMS to maintain the authenticity of the sidechain certificates from epoch to epoch. We now describe this inductive process in greater detail.

Initially, during the setup of the sidechain, $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$ is ran. Stakeholders generate their keys by invoking $(sk_i, vk_i) \leftarrow \mathsf{Gen}(\mathcal{P})$. In case $\mathsf{Gen}(\cdot)$ is a probabilistic algorithm, it is run in a derandomized fashion with its coins fixed to the output of a PRNG that is seeded by $H(\mathsf{ats_init}, \eta_{j_{\mathsf{start}}})$ where "ats_init" is a fixed label and H is a hash function. This ensures that $\mathcal{P}$ will be uniquely determined and will still be unpredictable. We note that this process is only suitable for ATMS that employ public-coin parameters; our ATMS constructions in Section 6.5 are only of this type.

For the induction base, $\mathcal{P}$ is published as part of the Genesis block $\overline{\mathsf{G}}$. Each time an $\mathbf{MC}$ stakeholder U_i posts the $\mathsf{sidechain_support}$ message to $\mathbf{MC}$, he also includes an ATMS key vk_i. Subsequently, when the $\mathbf{SC}$ is initialized, the stake distribution $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ is known to the $\mathbf{MC}$ participants. Hence, based on $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ and $\bar{\eta}_{j_{\mathsf{start}}}$, these can determine the last $2k$ slot leaders of epoch j_{start} in $\mathbf{SC}$, we will refer to them as the j_{start}-th *trust committee*. (In general, the j-th *trust committee* for $j \geq j_{\mathsf{start}}$ will be the set of last $2k$ slot leaders in epoch j.) $\mathbf{SC}$-maintainers (that also follow $\mathbf{MC}$) can also determine the j_{start}-th trust committee and therefore create $avk^{j_{\mathsf{start}}}$ from their public keys and insert it into the genesis block $\overline{\mathsf{G}}$ of $\mathbf{SC}$. They can also serve it as a special transaction to the $\mathbf{MC}$-maintainers to include into the mainchain. The correctness of $avk^{j_{\mathsf{start}}}$ can be readily verified by anyone following the mainchain using the procedure ACheck of the used ATMS.

For the induction step, consider an epoch $j > j_{\mathsf{start}}$ and assume that there exists an ATMS key of the previous epoch avk^{j-1}, known to the mainchain maintainers. Every honest $\mathbf{SC}$ slot leader among the last $2k$ slot leaders of $\mathbf{SC}$ epoch $j-1$ will produce a local signature s_i^j on the message $m = (j, \langle\mathsf{pending}_j\rangle, avk_j)$ using their private key sk_i^{j-1} by running $\mathsf{Sig}(sk_i^{j-1}, m)$, and include this signature into the block they create. The rest of the $\mathbf{SC}$ maintainers will verify that the epoch index, avk^j and $\langle\mathsf{pending}_j\rangle$ are correct (by ensuring $\mathsf{ACheck}(\mathcal{VK}^j, avk^j)$ is *true* for $\mathcal{VK}$ denoting the public keys of the last $2k$ slot leaders on $\mathbf{SC}$ for epoch j, and by recomputing the

Merkle tree commitment $\langle \mathsf{pending}_j \rangle$) and that s_i^j is valid by running $\mathsf{Ver}(m, vk_i^{j-1}, s_i^j)$, otherwise the block is considered invalid. Thanks to the chain growth property of the underlying Ouroboros protocol, after the last $2k$ slots of epoch $j-1$ the honest sidechain maintainers will all observe at least $k+1$ signatures among the $\{s_i^j : i \in [2k]\}$ desired ones. They then combine all of these local signatures into an aggregated ATMS signature $\sigma^j \leftarrow \mathsf{ASig}(m, \{(s_i^j, vk_i^{j-1})\}, \mathsf{keys}^j)$. This combined signature is then diffused as part of $\mathsf{sc_cert}_j$ on the mainchain network. The mainchain maintainers verify that it has been signed by the sidechain maintainers by checking that $\mathsf{AVer}(m, avk^{j-1}, \sigma^j)$ evaluates to *true* and include it in a mainchain block. This effectively hands over control to the new committee.

6.5 Constructing Ad-Hoc Threshold Multisignatures

In this section we give several ways to instantiate the ATMS primitive. We order them by increasing succinctness but also increasing complexity.

6.5.1 Plain ATMS

Given a EUF-CMA-secure signature scheme, combining signatures and keys can be implemented by plain concatenation. Subsequently, combined verification requires all signatures to be verified individually. This illustrates that the ATMS primitive is easy to realize if no concern is given to succinctness. The size of these aggregate signatures and aggregate keys is *quadratic* in the security parameter κ: for the aggregate key $2k$ individual keys of size κ bits each are concatenated (with $k = \Theta(\kappa)$), while the aggregate signature consists of at least $k+1$ individual signatures of size κ bits.

6.5.2 Multisignature-based ATMS

The previous construction can be improved by employing an appropriate multisignature scheme. In the construction below, we consider the multisignature scheme Π_{MGS} from [Bol03]. We make use of a homomorphic property of this scheme: any d individual signatures $\sigma_1, \ldots, \sigma_d$ created using secret keys belonging to (not necessarily unique) public keys $vk_1, \ldots, vk_d$ can be combined into a multisignature $\sigma = \prod_{i=1}^{d} \sigma_i$ that can then be verified using an aggregated public key $avk = \prod_{i=1}^{d} vk_i$.

Our multisignature-based t-ATMS construction works as follows: the procedures PGen, Gen, Sig and Ver work exactly as in Π_{MGS}. Given a set S, denote by $\langle S \rangle$ a Merkle-tree commitment to the set S created in some arbitrary, fixed, deterministic way. Procedure AKey, given a sequence of public keys $\mathcal{VK} = \{vk_i\}_{i=1}^{n}$ returns $avk = (\prod_{i=1}^{n} vk_i, \langle \mathcal{VK} \rangle)$. Since AKey is deterministic, $\mathsf{ACheck}(\mathcal{VK}, avk)$ simply recomputes it to verify avk. ASig takes the message m, d pairs of signatures with their respective public keys $\{\sigma_i, vk_i\}_{i=1}^{d}$ and $n-d$ additional public keys $\{\widehat{vk}_i\}_{i=1}^{n-d}$ and produces an aggregate signature

$$\sigma = \left(\prod_{i=1}^{d} \sigma_i, \{\widehat{vk}_i\}_{i=1}^{n-d}, \{\pi_{\widehat{vk}_i}\}_{i=1}^{n-d} \right) \tag{6.1}$$

where $\pi_{\widehat{vk}_i}$ denotes the (unique) inclusion proof of $\widehat{vk}_i$ in the Merkle commitment $\left\langle \{vk_i\}_{i=1}^{d} \cup \{\widehat{vk}_i\}_{i=1}^{n-d} \right\rangle$. Finally, the procedure AVer takes a message m, an aggregate key avk, and an aggregate signature σ parsed as in (6.1), and does the following: (a) verifies that each of the public keys $\widehat{vk}_i$ indeed belongs to a different leaf in the commitment $\langle \mathcal{VK} \rangle$ in avk using membership proofs $\pi_{\widehat{vk}_i}$; (b) computes avk' by dividing the first part of avk by $\prod_{i=1}^{n-d} \widehat{vk}_i$; (c) returns *true* if and only if $d \geq t$ and the first part of σ verifies as a Π_{MGS}-signature under avk'.

Note that the scheme Π_{MGS} requires vk_i to be accompanied by a (non-interactive) proof-of-possession (POP) [RY07] of the respective secret key. This POP can be appended to the public key and verified when the key is communicated in the protocol. For conciseness, we omit these proofs-of-knowledge from the description (but we include them in the size calculation below).

Asymptotic Complexity. This provides an improvement in our use case over the plain scheme: In the optimistic case where each of the $2k$ committee members create their local signatures, both the aggregate key avk and the aggregate signature σ are *linear* in the security parameter, which is optimal. If $r < k$ of the keys do *not* provide their local signatures, the construction falls back to being *quadratic* in the worst case if $r = k - 1$. However, for the practically relevant case where $r \ll k$ and almost all slot leaders produced a signature, this construction is clearly preferable.

Concrete space requirements. Concrete signature sizes in this scheme for practical parameters could be as follows. We set $k = 2160$ (as is done in the Cardano implementations of [KRDO17]) and for the signature of [Bol03] we have in bits: $|vk_i| = 272$, $|\sigma_i| = 528$ (N. Di Prima, V. Hanquez, personal communication, 16 Mar 2018), with $|vk_i + POP| = |vk_i| + |\sigma_i| = 800$ bits where POP represents the size of the proof of possession. Assuming 256-bit hash function is used for the Merkle tree construction, the size of the data which needs to be included in $\mathbf{MC}$ in the optimistic case during an epoch transition is $|avk| + |\sigma| + |\langle \mathsf{pending} \rangle| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| = 800 + 512 + 528 = 1840$ bits per epoch. In a case where 10% of participants fail to sign, the size will be $|avk| + |\sigma| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| + 0.1 \cdot 2 \cdot k(|vk_i + POP| + \log(k)|H(\cdot)) = 800 + 512 + 528 + 432 \cdot (500 + 12 \cdot 256) = 1544944$, or about 190 KB per epoch (which is approximately 5 days).

6.5.3 ATMS From Proofs of Knowledge

While the aggregate signatures construction seems sufficient for practice, it still requires a $\mathsf{sc_cert}$ transaction that is in the worst case quadratic in the security parameter. The approach below, based on proofs of knowledge, improves on that.

We define $avk \leftarrow \mathsf{AKey}(\mathcal{VK})$ to be the root of a Merkle tree that has $\mathcal{VK}$ at its leaves. Let $\mathsf{Sig}, \mathsf{Ver}$ come from any secure signature scheme. In our ATMS, the local signature is equal to $s_i = \mathsf{Sig}(sk_i, m)$, where sk_i is the secret key that corresponds to the vk_i verification key. Letting $S' = \{s_i\}$ be the signatures generated by a sequence $\mathcal{VK}'$ containing keys in $\mathcal{VK}$, the $\mathsf{ASig}(\mathcal{VK}, S', m)$ algorithm reconstructs the Merkle tree from $\mathcal{VK}$ and determines the membership proof π_i for each $vk_i \in \mathcal{VK}'$. Regarding the non-interactive argument of knowledge, the statement of interest is (avk, m) with witness $\{\pi_i, (s_i, vk_i)\}_{i \in S'}$ such that for all i we have that $\mathsf{Ver}(vk_i, m, s_i) = 1$ and π_i is a valid Merkle tree proof pointing to a unique leaf for every i. π_i demonstrates that vk_i is in avk. We also require $|S'| \geq t$. It is possible to construct succinct proofs for this statement using SNARKs [BCCT12] or even without any trusted setup using e.g., STARKs [BSBHR17] or Bulletproofs [BBB+18] in the Random Oracle model [BR93]. In both cases the actual size of the resulting signature will be at most logarithmic in k, while in the case of STARKs the verifier will also have time complexity logarithmic in k.

6.6 Security

In this section we give a formal argument establishing that the construction from Section 6.4 achieves pegging security of Definition 16.

6.6.1 Assumptions

Let $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ denote the honest-majority assumption for an Ouroboros ledger $\mathbf{L}$. Namely, $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ postulates that in all slots $t' \leq t$, the majority of stake in the stake distribution used to sample the slot leader for slot t' in $\mathbf{L}$ is controlled by honest parties (note that the distribution in question is SD and $\overline{\mathsf{SD}}^*$ for $\mathbf{MC}$ and $\mathbf{SC}$, respectively). Specifically, the adversary is restricted to $(1 - \epsilon)/2$ relative stake for some fixed $\epsilon > 0$.

The assumption $\mathbb{A}_{\mathbf{MC}}$ we consider for $\mathbf{MC}$ is precisely $\mathbb{A}_{\mathbf{MC}}[t] \triangleq \mathbb{A}_{\mathsf{hm}}(\mathbf{MC})[t]$, while the assumption $\mathbb{A}_{\mathbf{SC}}$ for $\mathbf{SC}$ is $\mathbb{A}_{\mathbf{SC}}[t] \triangleq \mathbb{A}_{\mathbf{MC}}[t] \wedge \mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$. The reason that $\mathbb{A}_{\mathbf{SC}}[t] \Rightarrow \mathbb{A}_{\mathbf{MC}}[t]$ is that $\mathbf{SC}$ uses merged

staking and hence cannot provide any security guarantees if the stake records on $\mathbf{MC}$ get corrupted. It is worth noting that it is possible to program $\mathbf{SC}$ to wean off $\mathbf{MC}$ and switch to independent staking; in such case the assumption for $\mathbf{SC}$ will transition to $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})$ (now with respect to $\overline{\mathsf{SD}}$) after the weaning slot and the two chains will become sidechains of each other.

Remark 1. We note that the assumption of honest majority in the distribution out of which leaders are *sampled* is one of two related ways of stating this requirement. The distribution from which sampling is performed corresponds to the actual stake distribution near the end of the previous epoch. Hence, the actual stake may have since shifted and may no longer be honest. Had we wanted to formulate this assumption in terms of the actual (current) stake distribution, we would have to state two different assumptions: (1) that the current actual stake has honest majority with some gap σ; and (2) that the rate of stake shifting is bounded by σ for the duration of (roughly) 2 epochs. From these two assumptions, one can conclude that the distribution from which leaders are elected is currently controlled by an honest majority. The latter approach was taken for example in [KRDO17].

6.6.2 Proof Overview

Proving our construction secure requires some case analysis. We summarize the intuition behind this endeavour before we proceed with the formal treatment.

The proof of Theorem 16 that shows that our construction from Section 6.4 has pegging security with overwhelming probability will be established as follows. We will borrow the fact that our construction achieves persistence and liveness from the original analysis [KRDO17] and state them as Lemma 9. The main challenge will be to establish the firewall property, which is done in Lemma 15. These properties together establish pegging security as required by Definition 16.

To show that the firewall property holds, we perform a case analysis, looking at the two cases of interest: when both $\mathbf{MC}$ and $\mathbf{SC}$ are secure (i.e., when $\mathbb{A}_{\mathbf{MC}} \wedge \mathbb{A}_{\mathbf{SC}}$ holds), and when only $\mathbf{MC}$ is secure while the security assumption of $\mathbf{SC}$ has been violated. As discussed above, the case where $\mathbf{SC}$ is secure and the security of $\mathbf{MC}$ has been violated cannot occur per definition of $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$, and so examining this case is not necessary.

First, we examine the case where both $\mathbf{MC}$ and $\mathbf{SC}$ are secure, but only concern ourselves with *direct observation* transactions, or transactions that can be verified without relying on sidechain certificates. We show that such transactions will always be correctly verified in this case.

Next, we establish that, when only $\mathbf{MC}$ is secure, it is impossible for the $\mathbf{MC}$ maintainers to accept a view inconsistent with the validity language, and hence the firewall property is maintained in the case of a sidechain failure.

Finally, the heart of the proof is a computational reduction (using the above partial results) showing how, given an adversary that breaks the firewall property, there must exist a receiving transaction on $\mathbf{MC}$ which breaks the validity of the scheme. Given such a transaction, we can construct an adversary against either the security of the underlying ATMS scheme or the collision resistance of the underlying hash function.

6.6.3 Liveness and Persistence

We begin by stating the persistence and liveness guarantees of our construction, they both follow directly from the guarantees shown for the standalone Ouroboros blockchain in [KRDO17].

Lemma 9 (Persistence and Liveness)**.** *Consider the construction of Section 6.4 with the assumptions* $\mathbb{A}_{\mathbf{SC}}, \mathbb{A}_{\mathbf{MC}}$. *For all slots t, if $\mathbb{A}_{\mathbf{SC}}[t]$ (resp. $\mathbb{A}_{\mathbf{MC}}[t]$) holds, then $\mathbf{SC}$ (resp. $\mathbf{MC}$) satisfies persistence and liveness up to slot t with overwhelming probability in k.*

We now restate the Common Prefix property of blockchains for future reference. If the Common Prefix property holds, then Persistence can be derived along the lines of [KRDO17].

Definition 20 (Common Prefix)**.** For every honest party P_1 and P_2 both maintaining the same ledger (i.e., either both maintaining $\mathbf{MC}$, or both maintaining $\mathbf{SC}$) and for every slot r_1 and r_2 such that $r_1 \leq r_2 \leq t$, let C_1 be the adopted chain of P_1 at slot r_1 and C_2 be the adopted chain of P_2 at slot r_2. The *k-common prefix property* for slot t states that $\mathsf{C}_2[: |\mathsf{C}_1[: -k]|] = \mathsf{C}_1[: -k]$.

6.6.4 The Firewall Property and MC-Receiving Transactions

Recall that the transactions in $\mathcal{T}_{\mathfrak{A}}$ can be partitioned into several classes with different validity-checking procedures. First, there are *local* transactions (where $\mathsf{send} = \mathsf{rec} = \mathsf{lid}$) and *sending* transactions (with $\mathsf{lid} = \mathsf{send} \neq \mathsf{rec}$). Then we have *receiving* transactions (with $\mathsf{send} \neq \mathsf{rec} = \mathsf{lid}$), which can be split into $\mathbf{SC}$*-receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{SC}$) and $\mathbf{MC}$*-receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{MC}$).

As the lemma below observes, if a transaction violates the firewall property in a certain situation, it must be a $\mathbf{MC}$-receiving transaction.

Lemma 10. *Consider an execution of the protocol of Section 6.4 at slot t in which* $\mathbf{MC}$ *and* $\mathbf{SC}$ *satisfy persistence. Suppose*

$$\mathsf{L} = \mathsf{merge}\left(\{\mathsf{L}^{\cup}_{\mathbf{MC}}[t], \mathsf{L}^{\cup}_{\mathbf{SC}}[t]\}\right) \notin \mathbb{V}_{\mathfrak{A}}$$

and suppose that $\mathcal{S}_t = \{\mathbf{SC}, \mathbf{MC}\}$*. Let* L' *be the minimum prefix of* L *such that* $\mathsf{L}' \notin \mathbb{V}_{\mathfrak{A}}$*. Then* $\mathsf{L}' \neq \varepsilon$ *and* $\mathsf{tx} \triangleq \mathsf{L}'[-1]$ *is an* $\mathbf{MC}$*-receiving transaction.*

Proof. The *base* property of the validity language implies $\mathsf{L}' \neq \varepsilon$, hence tx exists. Due to the minimality of L', Algorithm 2 returns *false* for L' but *true* for $\mathsf{L}'[: -1]$. Since it processes transactions sequentially, it must return *false* during the processing of tx. Suppose for contradiction that tx is not an $\mathbf{MC}$-receiving transaction; let us call such a transaction *direct* in this proof.

Algorithm 2 can output *false* while processing a direct transaction in the following cases: (a) in Line 17 when there is a Conservation Law violation; (b) in Line 8 when there is a signature validation failure; (c) in Line 13 when tx is a replay of a previous transaction; (d) in Line 22 when tx is a replay, or (e) in Line 27 when the pre-image transaction has not yet been processed. Hence, tx falls under one of these violations.

Due to persistence and the definition of $\mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, there exists an $\mathbf{MC}$ maintainer $P_{\mathbf{MC}}$ and an $\mathbf{SC}$ maintainer $P_{\mathbf{SC}}$, such that $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, respectively. Due to the *partitioning property* of merge, tx will be in $\mathsf{L}^{P_{\mathsf{lid}(\mathsf{tx})}}_{\mathsf{lid}(\mathsf{tx})}[t]$. We separately consider the two possibilities for $\mathsf{lid}(\mathsf{tx})$.

Case 1: $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$. In this case, the only violations that a direct tx can attain are (a), (b) and (c), as the cases (d) and (e) for $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$ do not pertain to a direct transaction. $P_{\mathbf{MC}}$ has reported $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ as its adopted state, hence $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint of $\textsc{verifytx}_{\mathbf{MC}}$ (as $\textsc{verifytx}_{\mathbf{MC}}$ checks for a fixpoint). The execution of $\textsc{verifytx}_{\mathbf{MC}}$ included every transaction in $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$. Therefore, $\textsc{verifytx}_{\mathbf{MC}}$ has accepted every transaction in every iteration until the last iteration, which processes tx. Consider, now, what happened in the last iteration of the execution of $\textsc{verifytx}_{\mathbf{MC}}$. In that iteration, $\textsc{verifytx}_{\mathbf{MC}}$ checks the validity of σ, the Conservation Law, and transaction replay. In all cases (a), (b) and (c), $\textsc{verifytx}_{\mathbf{MC}}$ will reject tx. But this could not have happened, as $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint, and we have a contradiction.

Case 2: $\mathsf{lid}(\mathsf{tx}) = \mathbf{SC}$. Let C_{mc} and C_{sc} be the $\mathbf{MC}$ and respectively $\mathbf{SC}$ chain adopted by $P_{\mathbf{SC}}$ at slot t (and recall that $P_{\mathbf{SC}}$ maintains both chains). Let $\mathsf{C}_{\mathsf{mc}}'$ be the chain adopted by $P_{\mathbf{MC}}$ at slot t. As before, $\textsc{annotatetx}_{\mathbf{SC}}(\mathsf{C}_{\mathsf{mc}}, \mathsf{C}_{\mathsf{sc}})$ must be a fixpoint of $\textsc{verifytx}_{\mathbf{SC}}$ (as $\textsc{verifytx}_{\mathbf{SC}}$ checks for a fixpoint). As in the previous case, tx cannot violate (a), (b), (c) and in this case nor (d), as this would constitute a fixpoint violation. Hence tx is an effect transaction and we will examine whether tx constitutes a violation of (e).

Let $\mathsf{tx}^{-1} \triangleq \mathsf{effect}^{-1}_{\mathbf{MC}\to\mathbf{SC}}(\mathsf{tx})$. Since tx is accepted by $\textsc{verifytx}_{\mathbf{SC}}$ on input $\textsc{annotatetx}_{\mathbf{SC}}(\mathsf{C}_{\mathsf{mc}}, \mathsf{C}_{\mathsf{sc}})$, we deduce that there exists some block $B \in \mathsf{C}_{\mathsf{mc}}[: -k]$ with $\mathsf{tx}^{-1} \in B$. But $\mathsf{C}_{\mathsf{mc}}'[: -k]$ is the longest stable chain among $\mathbf{MC}$ maintainers (due to $\mathsf{L}^{\cup}_{\mathbf{MC}}[t] = \mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$), hence $\mathsf{C}_{\mathsf{mc}}[: -k]$ is its prefix. Therefore $B \in \mathsf{C}_{\mathsf{mc}}'[: -k]$. Hence, $\mathsf{tx}^{-1} \in \mathbf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$. Due to the *partioning property* of merge, tx^{-1} must appear in the output of $\mathsf{merge}\left(\{\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t], \mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t]\}\right)$. Due to the *topological soundness* of merge, tx^{-1} must appear *before* tx in

$\mathsf{merge}\left(\{\mathsf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t], \mathsf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t]\}\right)$. Hence, it cannot be the case that (e) is violated, as the pre-image transaction exists. □ □

6.6.5 Firewall Property During Sidechain Failure

We now turn our attention to the case where the sidechain has suffered a "catastrophic failure" and so $\mathcal{S}_t = \{\mathbf{MC}\}$. We describe why a catastrophic failure in the sidechain does not violate the firewall property. To do this, we need to illustrate that, given a transaction sequence L which is accepted by the $\mathbf{MC}$ verifier, we can "fill in the gaps" with transactions from $\mathbf{SC}$ in order to produce a new transaction sequence $\vec{\mathsf{tx}}$ which is valid with respect to $\mathbb{V}_{\mathfrak{A}}$.

We prove this constructively in Lemma 12. The construction of such a sequence is described in Algorithm 13. The algorithm accepts a transaction sequence $\mathsf{L} \subseteq \mathcal{T}_{\mathbf{MC}}$ valid according to $\text{VERIFIER}_{\mathbf{MC}}$ and produces a transaction sequence $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$ satisfying $\pi_{\mathbf{MC}}(\vec{\mathsf{tx}}) = \mathsf{L}$, as desired.

The algorithm works by mapping each $\mathsf{tx} \in \mathsf{L}$ to one or more transactions in $\vec{\mathsf{tx}}$. The mapping is done by calling $\mathsf{plausibility\text{-}map}(\mathsf{tx})$ for each transaction individually. Hence each transaction in $\vec{\mathsf{tx}}$ has a specific preimage transaction in L, which can be shared by other transactions in $\vec{\mathsf{tx}}$. The mapping is performed as follows. If tx is a local transaction, then it is simply copied over, otherwise some extra transactions are included. Specifically, if it's an sending transaction tx, then first tx is included, and subsequently the funds are recovered by a corresponding transaction tx_1 on $\mathbf{SC}$, the effect transaction of tx. The funds are afterwards moved to a pool address pool_{pk} by a transaction tx_2. (Note that for this, we assume that the receiving account public key has a correspnding private key, as this key is needed to sign tx_2. As we are only demonstrating the existence of $\vec{\mathsf{tx}}$, Algorithm 13 does not need to be efficient and so assuming the existence of the private key is sufficient.) On the other hand, if it is an ($\mathbf{MC}$-)receiving transaction tx, the reverse procedure is followed. First, the funds are collected by tx_2 from the pool address pool_{pk} and moved into the $\mathbf{SC}$ address which will be used for the upcoming remote transaction. Then tx_1 moves the funds out of $\mathbf{SC}$ so that they can be collected by the corresponding tx on $\mathbf{MC}$. In the first case, the transaction sequence is $(\mathsf{tx}, \mathsf{tx}_1, \mathsf{tx}_2)$ and in the second case the sequence is $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx})$. Note that, in both cases, tx and tx_1 are identical, except for the fact that tx is recorded on $\mathbf{MC}$ while tx_1 is recorded on $\mathbf{SC}$; the latter is the effect (or pre-image, respectively) of the former.

The simple intuition behind this construction is that, in the plausible history $\vec{\mathsf{tx}}$ produced by Algorithm 13, the account pool_{pk} is holding all the money of the sidechain. More specifically, the balance that is maintained in the variable $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ is identical to the pool variable maintained by the $\mathbf{MC}$ verifier. This invariant is made formal in Lemma 11.

Lemma 11 (Plausible balances)**.** *Let $\mathsf{L} \in \mathcal{T}^*_{\mathfrak{A},\mathbf{MC}}$ and $\vec{\mathsf{tx}} \leftarrow \mathsf{plausible}(\mathsf{L})$. Consider an execution of Algorithm 2 on $\vec{\mathsf{tx}}$ and an execution of $\text{VERIFIER}_{\mathbf{MC}}$ on L. Let $\mathsf{tx} \in \mathsf{L}$. Call $\mathsf{pool}_{\mathsf{tx}}$ the value of the pool variable maintained by $\text{VERIFIER}_{\mathbf{MC}}$ prior to processing tx in its main* for *loop; call $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}}$ the value of the $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ variable prior to the iteration of its main* for *loop which processes the first item of $\mathsf{plausibility\text{-}map}(\mathsf{tx})$. For all $\mathsf{tx} \in \mathsf{L}$, the following invariant will hold: $\mathsf{pool}_{\mathsf{tx}} = \mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}}$.*

Proof. By direct inspection of the two algorithms, observe that the $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ are updated by Algorithm 2 only when $\mathsf{send}(\mathsf{tx}_a) \neq \mathsf{rec}(\mathsf{tx}_a)$. The balances are increased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{MC}$ (due to $\mathsf{tx}_2 \in \mathsf{plausibility\text{-}map}(\mathsf{tx})$ at Line 19 of Algorithm 13) and decreased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{SC}$ (due to $\mathsf{tx}_2 \in \mathsf{plausibility\text{-}map}(\mathsf{tx})$ at Line 25 of Algorithm 13). Exactly the same accounting is performed by $\text{VERIFIER}_{\mathbf{MC}}$ when the respective tx is processed. □ □

We now prove the correctness of Algorithm 13 in Lemma 12.

Lemma 12 (Plausibility)**.** *For all $\mathsf{L} \in \mathcal{T}^*_{\mathfrak{A},\mathbf{MC}}$, if $\text{VERIFYTX}_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ then $\vec{\mathsf{tx}} \leftarrow \mathsf{plausible}(\mathsf{L})$ will satisfy $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$.*

Algorithm 13 The plausible transaction sequence generator.

1: $(\mathsf{pool}_{sk}, \mathsf{pool}_{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$
2: **function** $\mathsf{plausible}(L)$
3: $\quad \vec{\mathsf{tx}} \leftarrow \varepsilon$
4: $\quad$ **for** $\mathsf{tx} \in L$ **do**
5: $\quad\quad \vec{\mathsf{tx}} \leftarrow \vec{\mathsf{tx}} \,\|\, \mathsf{plausibility\text{-}map}(\mathsf{tx})$
6: $\quad$ **end for**
7: $\quad$ **return** $\vec{\mathsf{tx}}$
8: **end function**
9: **function** $\mathsf{plausibility\text{-}map}(\mathsf{tx})$
10: $\quad \triangleright$ *Destructure* tx *into its constituents*
11: $\quad (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma) \leftarrow \mathsf{tx}$
12: $\quad$ **if** $\mathsf{send} = \mathsf{rec}$ **then**
13: $\quad\quad$ **return** (tx)
14: $\quad$ **end if**
15: $\quad$ **if** $\mathsf{send} = \mathbf{MC}$ **then**
16: $\quad\quad \mathsf{tx}_1 \leftarrow \mathsf{effect}_{\mathbf{MC}\to\mathbf{SC}}(\mathsf{tx})$
17: $\quad\quad$ Construct a valid σ_2 using the private key corresponding to rAcc
18: $\quad\quad$ Generate a fresh txid_2
19: $\quad\quad \mathsf{tx}_2 \leftarrow \big(\mathsf{txid}_2, \mathbf{SC}, (\mathbf{SC}, \mathsf{rAcc}), (\mathbf{SC}, \mathsf{pool}_{pk}), v, \sigma_2\big)$
20: $\quad\quad$ **return** $(\mathsf{tx}, \mathsf{tx}_1, \mathsf{tx}_2)$
21: $\quad$ **end if**
22: $\quad$ **if** $\mathsf{send} = \mathbf{SC}$ **then**
23: $\quad\quad$ Construct a valid σ_2 using pool_{sk}
24: $\quad\quad$ Generate a fresh txid_2
25: $\quad\quad \mathsf{tx}_2 \leftarrow \big(\mathsf{txid}_2, \mathbf{SC}, (\mathbf{SC}, \mathsf{pool}_{pk}), (\mathbf{SC}, \mathsf{sAcc}), v, \sigma_2\big)$
26: $\quad\quad \mathsf{tx}_1 \leftarrow \mathsf{effect}^{-1}_{\mathbf{SC}\to\mathbf{MC}}(\mathsf{tx})$
27: $\quad\quad$ **return** $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx})$
28: $\quad$ **end if**
29: **end function**

Proof. Suppose for contradiction that $\vec{\mathsf{tx}} \notin \mathbb{V}_{\mathfrak{A}}$ and let $\vec{\mathsf{tx}}'$ be the minimum prefix of $\vec{\mathsf{tx}}$ such that $\vec{\mathsf{tx}}' \notin \mathbb{V}_{\mathfrak{A}}$. From the validity language *base* property we have that $\vec{\mathsf{tx}}' \neq \varepsilon$ and so it must have at least one element. Let $\mathsf{tx} \triangleq \vec{\mathsf{tx}}'[-1]$ and let $\mathsf{tx}_\mathsf{L} \in \mathsf{L}$ be the input to $\mathsf{plausibility\text{-}map}$ which caused tx to be included in $\vec{\mathsf{tx}}$ in the execution of $\mathsf{plausible}$ in Algorithm 13. Since Algorithm 2 processes transactions sequentially, and by the minimality of $\vec{\mathsf{tx}}'$, it must return *false* when tx is processed.

We distinguish the following cases for tx_L:

Case 1: Local transaction: $\mathsf{send}(\mathsf{tx}_\mathsf{L}) = \mathsf{rec}(\mathsf{tx}_\mathsf{L})$. Then $\mathsf{tx} = \mathsf{tx}_\mathsf{L}$ and $\mathsf{send}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx})$. Since L is a fixpoint of $\text{VERIFYTX}_{\mathbf{MC}}$, tx must (a) have a valid signature σ, (b) not be a replay transaction, and (c) respect the Conservation Law. As tx_L is a local transaction satisfying all of (a), (b) and (c), therefore $\vec{\mathsf{tx}}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

Case 2: Sending transaction: $\mathsf{send}(\mathsf{tx}_\mathsf{L}) = \mathbf{MC}$ and $\mathsf{rec}(\mathsf{tx}_\mathsf{L}) = \mathbf{SC}$. In this case, let $(\mathsf{tx}_\mathsf{L}, \mathsf{tx}_1, \mathsf{tx}_2) = \mathsf{plausibility\text{-}map}(\mathsf{tx}_\mathsf{L})$. If $\mathsf{tx} = \mathsf{tx}_\mathsf{L}$, then tx is a sending transaction and we can apply the same reasoning to argue that it will respect properties (a), (b) and (c). But those are the only violations for which Algorithm 2 can reject an sending transaction, and hence $\vec{\mathsf{tx}}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

If $\mathsf{tx} = \mathsf{tx}_1$, then Algorithm 2 must return *true*. To see this, consider the cases when Algorithm 2 returns *false*: (d) a replay failure in Line 22, which cannot occur as tx_L has been accepted by $\text{VERIFYTX}_{\mathbf{MC}}$ and so $\text{VERIFYTX}_{\mathbf{MC}}$ must have *seen* tx_L only once while Algorithm 2 must be seeing it for exactly the second time; or

(e) a mismatch failure in Line 22 which cannot occur as tx_1 is constructed identical to tx_L.

If $\mathsf{tx} = \mathsf{tx}_2$ then $\mathsf{send}(\mathsf{tx}) = \mathsf{rec}(\mathsf{tx})$. This transaction cannot cause Algorithm 2 to return *false*. To see this, consider the cases when Algorithm 2 returns *false*: (a) a signature failure in Line 8 cannot occur because σ_2 was constructed correctly and the signature scheme is correct; (b) a replay failure in Line 13 cannot occur because txid_2 is fresh; (c) a conservation failure in Line 17 cannot occur because the immediately preceding transaction $\vec{\mathsf{tx}}'[-2]$ supplies sufficient balance.

Case 3: Receiving transaction: $\mathsf{send}(\mathsf{tx}_\mathsf{L}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}_\mathsf{L}) = \mathbf{MC}$. In this case, let $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx}_\mathsf{L}) =$ $\mathsf{plausibility\text{-}map}(\mathsf{tx}_\mathsf{L})$. The argument for $\mathsf{tx} = \mathsf{tx}_\mathsf{L}$ and $\mathsf{tx} = \mathsf{tx}_1$ is as in *Case 2*. For the case of $\mathsf{tx} =$ tx_2, the same argument as before holds for a signature validity and for replay protection. It suffices to show that the conservation law is not violated. This is established in Lemma 11 by the invariant that $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} =$ $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}}$ that holds prior to processing tx_2, as it is the first transaction of a triplet produced by $\mathsf{plausibility\text{-}map}$. As $\textsc{verifytx}_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ then therefore $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} - v \geq 0$ and so $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}} -$ $amount \geq 0$ and Algorithm 2 returns *true*.

All three cases result in a contradiction, concluding the proof. □ □

Lemma 13 ($\mathbf{SC}$ failure firewall)**.** *Consider any execution of the construction of Section 6.4 in which persistence holds for* $\mathbf{MC}$*. For all slots t such that $\mathcal{S}_t = \{\mathbf{MC}\}$ we have that*

$$\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}}) \ .$$

Proof. From the assumption that persistence holds, there exists some $\mathbf{MC}$ party P for which $\mathbf{L}^{P}_{\mathbf{MC}}[t] =$ $\mathbf{L}^{\cup}_{\mathbf{MC}}[t]$. Additionally, $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) = \mathbf{L}^{\cup}_{\mathbf{MC}}[t]$ due to the *partitioning* property. It suffices to show that there exists some $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$ such that $\pi_{\{\mathbf{MC}\}}(\vec{\mathsf{tx}}) = \mathbf{L}^{P}_{\mathbf{MC}}[t]$. Let $\vec{\mathsf{tx}} \leftarrow \mathsf{plausible}(\mathbf{L}^{P}_{\mathbf{MC}}[t])$. We have $\textsc{verifier}_{\mathbf{MC}}(\mathbf{L}^{P}_{\mathbf{MC}}[t]) = true$, so apply Lemma 12 to obtain that $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$.

To see that $\pi_{\{\mathbf{MC}\}}(\vec{\mathsf{tx}}) = \mathbf{L}^{P}_{\mathbf{MC}}[t]$, note that Algorithm 13 for input L includes all $\mathsf{tx} \in \mathsf{L}$ in the same order as in its input. Furthermore, all $\mathsf{tx} \in \vec{\mathsf{tx}}$ such that $\mathsf{tx} \notin \mathsf{L}$ have $\mathsf{lid}(\mathsf{tx}) = \mathbf{SC}$ and so are excluded from the projection. □ □

6.6.6 General Firewall Property

In preparation for establishing the full firewall property, we state the following simple technical lemma.

Lemma 14 (Honest subsequence)**.** *Consider any set S of $2k$ consecutive slots prior to slot t in an execution of an Ouroboros ledger $\mathbf{L}$ such that $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ holds. Then $k+1$ slots of S are honest, except with negligible probability.*

Proof sketch. If the adversary controlled at least k out of any $2k$ consecutive slots, he could use them to produce an alternative k-blocks long chain for this interval without any help from the honest parties, resulting in a violation of common prefix and hence persistence (cf. Lemma 9). □ □

We are now ready to prove our key lemma, showing that our construction satisfies the firewall property.

Lemma 15 (Firewall)**.** *For all PPT adversaries $\mathcal{A}$, the construction of Section 6.4 with a secure ATMS and a collision-resistant hash function satisfies the firewall property with respect to assumptions $\mathbb{A}_{\mathbf{MC}}, \mathbb{A}_{\mathbf{SC}}$ with overwhelming probability in k.*

Proof. Let $\mathcal{A}$ be an arbitrary PPT adversary against the firewall property, and $\mathcal{Z}$ be an arbitrary environment for the execution of $\mathcal{A}$. We will construct the following PPT adversaries:

1. $\mathcal{A}_1$ is an adversary against ATMS.
2. $\mathcal{A}_2$ is a collision adversary against the hash function.

We first describe the construction of these adversaries.

The adversary $\mathcal{A}_1$. $\mathcal{A}_1$ simulates the execution of $\mathcal{A}$ and $\mathcal{Z}$ and of two populations of maintainers for two blockchains, $\mathbf{MC}$ and $\mathbf{SC}$, which run the protocol Π (either the $\mathbf{MC}$ or the $\mathbf{SC}$-maintainer part respectively) and spawns parties according to the mandates of the environment $\mathcal{Z}$ as follows. For all parties that are spawned as $\mathbf{MC}$ maintainers, $\mathcal{A}_1$ generates keys internally by invoking the Gen algorithm of the ATMS scheme. For all parties that are spawned as $\mathbf{SC}$ maintainers, $\mathcal{A}_1$ uses the oracle $\mathcal{O}^{\mathsf{gen}}$ to produce the public keys vk_i.

Whenever $\mathcal{A}$ requests that a (block or transaction) signature in $\mathbf{SC}$ is created, $\mathcal{A}_1$ invokes its oracle $\mathcal{O}^{\mathsf{sig}}$ to obtain the respective signature to provide to $\mathcal{A}$. When $\mathcal{A}$ requests that a $\mathbf{MC}$ signature is created, $\mathcal{A}_1$ uses its own generated private key to sign by invoking the Sig algorithm of the ATMS scheme. If $\mathcal{A}$ requests the corruption of a certain party P^*, then $\mathcal{A}_1$ reveals P^*'s private key to $\mathcal{A}$ as follows: If P^* is a $\mathbf{MC}$ maintainer, then the secret key is directly available to $\mathcal{A}_1$, so it is immediately returned. Otherwise, if P^* is a $\mathbf{SC}$ maintainer, then $\mathcal{A}_1$ obtains the secret key of P^* by invoking the oracle $\mathcal{O}^{\mathsf{cor}}$.

For every time slot t of the execution, $\mathcal{A}_1$ inspects all pairs $(P_{\mathbf{MC}}, P_{\mathbf{SC}})$ of honest parties such that $P_{\mathbf{MC}}$ is a $\mathbf{MC}$ maintainer and $P_{\mathbf{SC}}$ is a $\mathbf{SC}$ maintainer such that $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$ (if such parties exist). Let $\mathsf{L}_1 = \mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t]$ and $\mathsf{L}_2 = \mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t]$. The adversary obtains the stable portion of the honestly adopted chain, namely $\mathsf{C}_1 = \mathsf{C}^{P_{\mathbf{MC}}}[t][: -k]$ and the transactions included in C_1, namely L'_1 (note that $\mathsf{L}'_1 \neq \mathsf{L}_1$ if L'_1 contains certificate transactions). $\mathcal{A}_1$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{A}}$, to deduce whether $\mathcal{A}$ has succeeded. Note that both the evaluation of merge on arbitrary states and the verification of inclusion in $\mathbb{V}_{\mathfrak{A}}$ are efficiently computable and hence $\mathcal{A}_1$ can execute them. If $\mathcal{A}_1$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns FAILURE (in the latter part of this proof, we will argue that all $\mathcal{A}_1$ failures occur with negligible probability conditioned on the event that $\mathcal{A}$ is successful, unless $\mathcal{A}_2$ is successful).

Otherwise it obtains the minimum t for which this holds and the L for this t. Because of the *base property* of the validity language, we have that $\epsilon \in \mathbb{V}_{\mathfrak{A}}$ and therefore $\mathsf{L} \neq \epsilon$. Let L^* be the minimum prefix of L such that $\mathsf{L}^* \notin \mathbb{V}_{\mathfrak{A}}$ and let $\mathsf{tx} = \mathsf{L}^*[-1]$. If tx has $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_1$ returns FAILURE. Now therefore $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$ (and so $\mathsf{tx} \in \mathsf{L}_1$). Hence, tx references a certain certificate transaction, say tx'. Due to the algorithm executed by $\mathbf{MC}$ maintainers for validation, we will have that $\mathsf{tx}' \in \mathsf{L}'_1\{: \mathsf{tx}\}$.

Let $\vec{\mathsf{tx}}^*$ be the subsequence of L'_1 containing all certificate transactions up to and including tx'. We will argue that there must exist some ATMS forgery among one of the certificate transactions in $\vec{\mathsf{tx}}^*$. $\mathcal{A}_1$ looks at every transaction $\mathsf{sc_cert}_j \in \vec{\mathsf{tx}}^*$ (and note that it will correspond to a unique epoch e_j). $\mathsf{sc_cert}_j$ contains a message $m = (j, \langle \mathsf{pending}_j \rangle, avk^j)$ and a signature σ_j. $\mathcal{A}_1$ extracts the epoch e_j in which $\mathsf{sc_cert}_j$ was confirmed in C_1 (and note that we must have $j > 0$). $\mathcal{A}_1$ collects the public keys elected for the last $2k$ slots of epoch e_{j-1} according to the view of $P_{\mathbf{SC}}$ into a set keys_{j-1} and similarly for keys_j. $\mathcal{A}_1$ collects the pending cross-chain transactions of e_{j-1} according to the view of $P_{\mathbf{SC}}$ into $\mathsf{pending}'_j$, and creates the respective Merkle-tree commitment $\langle \mathsf{pending}'_j \rangle$. $\mathcal{A}_1$ checks whether the following *certificate violation* condition holds:

$$\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \wedge \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1}) \wedge \left(\neg \mathsf{ACheck}(\mathsf{keys}_j, avk^j) \vee \langle \mathsf{pending}_j \rangle \neq \langle \mathsf{pending}'_j \rangle\right) \tag{6.2}$$

where avk^{j-1} is extracted from $\mathsf{sc_cert}_{j-1}$ according to the view of $P_{\mathbf{SC}}$, unless $j = 1$ in which case avk^0 is known. If the condition (6.2) holds for no j then $\mathcal{A}_1$ returns FAILURE, otherwise it denotes by j^* the minimum j for which (6.2) holds and outputs the tuple $(m, \sigma_{j^*}, avk^{j^*-1}, \mathsf{keys}^{j^*-1})$.

The adversary $\mathcal{A}_2$. Like $\mathcal{A}_1$, $\mathcal{A}_2$ simulates the execution of $\mathcal{A}$ including two populations of maintainers and spawns parties according to the mandates of the environment $\mathcal{Z}$. For *all* these parties, $\mathcal{A}_2$ generates keys internally. When $\mathcal{A}$ requests that a transaction is created, $\mathcal{A}_2$ provides the signature with its respective private key. If $\mathcal{A}$ requests the corruption of a certain party, say P^*, then $\mathcal{A}_2$ provides the respective private key to $\mathcal{A}$.

For every time slot t of the execution, $\mathcal{A}_2$ inspects all pairs of honest parties such that $P_{\mathbf{MC}}$ is a $\mathbf{MC}$ maintainer and $P_{\mathbf{SC}}$ is a $\mathbf{SC}$ maintainer such that $\mathsf{L}^{P_{\mathbf{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathbf{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$ and obtains the variables $\mathsf{L}_1, \mathsf{L}_2, \mathsf{C}_1, \mathsf{L}'_1$ as before. $\mathcal{A}_2$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{A}}$, to deduce whether $\mathcal{A}$ has succeeded. If $\mathcal{A}_2$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns FAILURE. Let tx be as

in $\mathcal{A}_1$. If $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_2$ returns FAILURE. Then tx references a certain certificate transaction $\mathsf{sc_cert}_j = (j, \langle \mathsf{pending}_j \rangle, avk^j, \sigma_j)$ and uses a Merkle tree proof π which proves the inclusion of tx in $\mathsf{pending}_j$. If $\mathsf{sc_cert}_j \notin \mathsf{L}'_1$, then $\mathcal{A}_2$ returns FAILURE. When $\mathsf{sc_cert}_j$ was accepted by $P_{\mathbf{SC}}$, $\mathsf{pending}_j$ included a set of transactions $\vec{\mathsf{tx}}$ in the view of $P_{\mathbf{SC}}$. If $\mathsf{tx} \in \vec{\mathsf{tx}}$, then $\mathcal{A}_2$ returns FAILURE. Otherwise, the Merkle tree $\langle \mathsf{pending}_j \rangle$ was constructed from $\vec{\mathsf{tx}}$, but a proof-of-inclusion π for $\mathsf{tx} \notin \vec{\mathsf{tx}}$ was created. From this proof, $\mathcal{A}_2$ extracts a hash collision and returns it.

Probability analysis. Define the following events:

- SC-FORGE$[t]$: $\mathcal{A}$ is successful at slot t, i.e., $\pi_{\mathfrak{A}}\left(\mathsf{merge}(\{\mathsf{L}_i^{\cup}[t] : i \in \mathcal{S}_t\})\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$.
- ATMS-FORGE: $\mathcal{A}_1$ finds an index j^* for which the condition (6.2) occurs.
- HASH-COLLISION: $\mathcal{A}_2$ finds a hash function collision.

Note that ledger states in the protocol only contain $\mathfrak{A}$-transactions, hence $\pi_{\mathfrak{A}}$ is the identity function and SC-FORGE$[t]$ is equivalent to $\mathsf{merge}\left(\{\mathbf{L}_i^{\cup}[t] : i \in \mathcal{S}_t\}\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$. We will now show that for every t, the probability $\Pr[\text{SC-FORGE}[t]]$ is negligible. We distinguish two cases:

Case 1: $\mathcal{S}_t = \{\mathbf{MC}, \mathbf{SC}\}$. In this case Persistence holds for both $\mathbf{MC}$ and $\mathbf{SC}$, and $\pi_{\mathcal{S}_t}$ is the identity function. We deal with this case in two successive claims (both implicitly conditioning on being in Case 1). First we show that, if SC-FORGE$[t]$ occurs, then one of ATMS-FORGE, HASH-COLLISION occurs. Therefore applying a union bound, we will have that:

$$\Pr[\text{SC-FORGE}[t]] \leq \Pr[\text{ATMS-FORGE}] + \Pr[\text{HASH-COLLISION}] \ .$$

Second, we show that $\Pr[\text{ATMS-FORGE}]$ is negligible (and the negligibility of $\Pr[\text{HASH-COLLISION}]$ follows from our assumption that the hash function is collision resistant).

Claim 1a: SC-FORGE$[t] \Rightarrow$ ATMS-FORGE $\vee$ HASH-COLLISION.
Because persistence holds in both $\mathbf{MC}$ and $\mathbf{SC}$, we know that there exist two parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$ such that at slot t we have that $\mathbf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t] = \mathbf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathbf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t] = \mathbf{L}_{\mathbf{SC}}^{\cup}[t]$, respectively. Therefore SC-FORGE$[t]$ implies

$$\mathsf{merge}(\{\mathbf{L}_{\mathbf{MC}}^{P_{\mathbf{MC}}}[t], \mathbf{L}_{\mathbf{SC}}^{P_{\mathbf{SC}}}[t]\}) \notin \mathbb{V}_{\mathfrak{A}} \ .$$

Let $\mathsf{tx}, \mathsf{tx}'$ be as in the definition of $\mathcal{A}_1$. By Lemma 10 and using $\mathbf{MC}$ and $\mathbf{SC}$ persistence, tx will exist and be an $\mathbf{MC}$-receiving transaction. Hence, $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$. Therefore, tx' will also exist. If $\mathcal{A}_1$ finds the index j^* for which (6.2) is satisfied, then ATMS-FORGE has occured and the claim is established, so let us assume otherwise. Hence, for each certificate $\mathsf{sc_cert}_j$ containing a message $m = (j, \langle \mathsf{pending}_j \rangle, avk^j)$, it holds that

$$\left(\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \wedge \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})\right) \Rightarrow \left(\mathsf{ACheck}(\mathsf{keys}_j, avk^j) \wedge \langle \mathsf{pending}_j \rangle = \langle \mathsf{pending}'_j \rangle\right) \ . \tag{6.3}$$

Therefore, we have a chain of certificates, each of which is signed with a valid key avk^{j-1} and attests to the validity of the next key avk^j. For all of these certificates, $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ holds, as it has been verified by $P_{\mathbf{MC}}$. Furthermore, by an induction argument (where the base case comes from the construction of avk^0 and the induction step follows from (6.3)) we have $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$ as well.

As tx' is a certificate transaction which appears last in the above chain (with some index $\mathsf{sc_cert}_k$), the above implication also holds for tx', and so does its premise $\mathsf{AVer}(m, avk^{k-1}, \sigma_k) \wedge \mathsf{ACheck}(\mathsf{keys}_{k-1}, avk^{k-1})$. Therefore, the conclusion of the implication $\langle \mathsf{pending}_k \rangle = \langle \mathsf{pending}'_k \rangle$ holds. However, the sending transaction corresponding to tx has been proven to belong to the Merkle Tree $\langle \mathsf{pending}_k \rangle$ (as verified by $P_{\mathbf{MC}}$), but does not belong to $\mathsf{pending}'_k$ (by the selection of tx). This constitutes a Merkle Tree collision, which translates to a hash collision. The construction of $\mathcal{A}_2$ outputs exactly this collision, and in this case we deduce that $\mathcal{A}_2$ is successful and HASH-COLLISION follows.

Claim 1b: $\Pr[\text{ATMS-FORGE}]$ is negligible.
Suppose that ATMS-FORGE occurs. We will argue that, in this case, $\mathcal{A}_1$ will have computed an ATMS forgery, which is a negligible event by the assumption that the used ATMS is secure.

From the assumption that ATMS-FORGE has occurred, at epoch e_j we have that $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ and $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$, but $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle\mathsf{pending}_j\rangle \neq \langle\mathsf{pending}'_j\rangle$. From Lemma 14 and using $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$, we deduce that in the last $2k$ slots of epoch e_{j-1}, at least $k+1$ must be honest. Since e_j is the earliest epoch in which this occurs, this means that keys_{j-1} corresponds to the last $2k$ slot leaders of epoch e_{j-1}, and all honest parties agree on the same $2k$ slot leaders. Hence, in the ATMS game, the number of keys in keys corrupted by the adversary through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$ is less than k. Furthermore, since $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle\mathsf{pending}_j\rangle \neq \langle\mathsf{pending}'_j\rangle$, the message m contains either an invalid future aggregate key, an invalid Merkle Tree root of outgoing cross-chain transactions, or both. Hence, no honest party will sign the message m for this epoch and therefore $|Q^{\mathsf{sig}}[m]| = 0$. Hence $q < k$, and $\mathcal{A}_1$ wins the ATMS security game.

Case 2: $\mathcal{S}_t \neq \{\mathbf{MC}, \mathbf{SC}\}$. If $\mathbf{MC} \notin \mathcal{S}_t$ then, since $\mathbb{A}_{\mathbf{MC}}[t] \Rightarrow \mathbb{A}_{\mathbf{SC}}[t]$, we have $\mathcal{S}_t = \emptyset$ and $\neg\text{SC-FORGE}[t]$, as $\epsilon \in \mathbb{V}_{\mathfrak{A}}$ by the *base property*. It remains to consider the case $\mathcal{S}_t = \{\mathbf{MC}\}$. Using $\mathbf{MC}$ persistence, by Lemma 13 we obtain $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}})$, and hence SC-FORGE$[t]$ did not occur.

From the two above cases, we conclude that for every t, $\Pr[\text{SC-FORGE}[t]] \leq \mathsf{negl}$. As the total number of slots is polynomial, we have shown that with overwhelming probability, we have that for all slots t and for all $\mathsf{A} \in \bigcup_{i \in \mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$, $\pi_{\mathsf{A}}\left(\mathsf{merge}\left(\{\mathbf{L}^{\cup}_i[t] \,:\, i \in \mathcal{S}_t\}\right)\right) \in \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathsf{A}})$, concluding the proof. □ □

Lemmas 9 and 15 together directly imply the following theorem.

Theorem 16 (Pegging Security)**.** *Consider the synchronous setting as defined in Section 6.2.1 with $2R$-semiadaptive corruptions as defined in Section 6.2.1. The construction of Section 6.4 using a secure ATMS and a collision resistant hash function is pegging secure with liveness parameter $u = 2k$ with respect to assumptions $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$ defined above, and* merge*,* effect *and $\mathbb{V}_{\mathfrak{A}}$ defined in Section 6.4.2.*

Supplementary Material

6.7 The Diffuse Functionality

In the model described in Section 6.2.1 we employ the "Delayed Diffuse" functionality of [DGKR18], which we now describe in detail for completeness. The functionality is parameterized by $\Delta \in \mathbb{N}$ and denoted $\mathsf{DDiffuse}_{\Delta}$. The functionality executes one step (round) per slot. $\mathsf{DDiffuse}_{\Delta}$ interacts with the environment $\mathcal{Z}$, stakeholders $U_1, \ldots, U_n$ and adversary $\mathcal{A}$, working as follows for each round: $\mathsf{DDiffuse}_{\Delta}$ maintains an incoming string for each party P_i that participates. A party, if activated, can fetch the contents of its incoming string, hence it behaves as a mailbox. Furthermore, parties can give an instruction to the functionality to diffuse a message. Activated parties can diffuse once per round.

When the adversary $\mathcal{A}$ is activated, it can: (a) read all inboxes and all diffuse requests and deliver messages to the inboxes in any order; (b) for any message m obtained via a diffuse request and any party P_i, $\mathcal{A}$ may move m into a special string $\mathsf{delayed}_i$ instead of the inbox of P_i. $\mathcal{A}$ can decide this individually for each message and each party; (c) for any party P_i, $\mathcal{A}$ can move any message from the string $\mathsf{delayed}_i$ to the inbox of P_i.

At the end of each round, the functionality ensures that every message that was either (a) diffused in this round and not put to the string $\mathsf{delayed}_i$ or (b) removed from the string $\mathsf{delayed}_i$ in this round is delivered to the inbox of party P_i. If a message currently present in $\mathsf{delayed}_i$ was originally diffused Δ slots ago, the functionality removes it from $\mathsf{delayed}_i$ and appends it to the inbox of party P_i.

Upon receiving $(\mathsf{Create}, U, \mathcal{C})$ from the environment, the functionality spawns a new stakeholder with chain $\mathcal{C}$ as its initial local chain (as in [KRDO17, DGKR18]).

6.8 Adaptation to Other Proof-of-Stake Blockchains

Our construction can be adapted to work with other provably secure proof-of-stake blockchains discussed in Section 6.2.3: Ouroboros Praos [DGKR18], Ouroboros Genesis [BGK$^+$18], Snow White [BPS16], and Algorand [Mic16]. Here we assume some familiarity with the considered protocols and refer the interested reader to the original papers for details.

6.8.1 Ouroboros Praos and Ouroboros Genesis

These protocols [DGKR18, BGK$^+$18] are strongly related and differ from each other only in the chain-selection rule they use, which is irrelevant for our discussion here, hence we consider both of the protocols simultaneously. Ouroboros Praos was shown secure in the semi-synchronous model with fully adaptive corruptions (cf. Section 6.2.1) and this result extends to Ouroboros Genesis. Despite sharing the basic structure with Ouroboros, they differ in several significant points which we now outline.

The slot leaders are elected differently: Namely, each party for each slot evaluates a verifiable random function (VRF, [DY05]) using the secret key associated with their stake, and providing as inputs to the VRF both the slot index and the epoch randomness. If the VRF output is below a certain threshold that depends on the party's stake, then the party is an eligible slot leader for that slot, with the same consequences as in Ouroboros. Each leader then includes into the block it creates the VRF output and a proof of its validity to certify her eligibility to act as slot leader. The probability of becoming a slot leader is roughly proportional to the amount of stake the party controls, however now it is independent for each slot and each party, as it is evaluated locally by each stakeholder for herself. This local nature of the leader election implies that there will inevitably be some slots with no, or several, slot leaders. In each epoch j, the stake distribution used in Praos and Genesis for slot leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $j-2$. Additionally, the *epoch randomness* η_j for epoch j is derived as a hash of additional VRF-values included into blocks from the first two thirds of epoch $j-1$ for this purpose by the respective slot leaders. Finally, the protocols use *key-evolving signatures* for block signing, and in each slot the honest parties are mandated to update their private key, contributing to their resilience to adaptive corruptions.

Ouroboros Praos was shown [DGKR18] to achieve persistence and liveness under weaker assumptions than Ouroboros, namely: (1) Δ-semi-synchronous communication (where Δ affects the security bounds but is unknown to the protocol); (2) majority of the stake is always controlled by honest parties. In particular, Ouroboros Praos is secure in face of fully adaptive corruptions without any corruption delay. Ouroboros Genesis provides the same guarantees as Praos, as well as several other features that will not be relevant for our present discusion.

Construction of Pegged Ledgers. The main difference compared to our treatment of Ouroboros would be in the construction of the sidechain certificate (cf. Section 6.4.3). The need for a modification is caused by the private, local leader selection using VRFs in these protocols, which makes it impossible to identify the set of slot leaders for the suffix of an epoch at the beginning of this epoch, as done for Ouroboros.

The sidechain certificate included in $\mathbf{MC}$ at the beginning of epoch j would hence contain the following, for parameters Q and T specified below:

1. the epoch index;
2. a Merkle commitment to the list of withdrawals as in the case of Ouroboros;
3. a Merkle commitment to the $\mathbf{SC}$ stake distribution $\overline{\mathsf{SD}}_j$;
4. a list of Q public keys;
5. Q inclusion proofs (with respect to $\overline{\mathsf{SD}}_{j-1}$ contained in the previous certificate) and Q VRF-proofs certifying that these Q keys belong to slot leaders of Q out of the last T slots in epoch $j-1$;
6. Q signatures from the above Q public keys on the above; these can be replaced by a single aggregate signature to save space on $\mathbf{MC}$.

The parameters Q and T have to be chosen in such a way that with overwhelming probability, there will be a chain growth of at least Q blocks during the last T slots of epoch $j-1$, but the adversary controls Q slots in this period only with negligible probability (and hence at least one of the signatures will have to come from an honest slot leader). The existence of such constants for $T = \Theta(k)$ was shown in [BGK+18].

While the above sidechain certificate is larger (and hence takes more space on $\mathbf{MC}$) than the one we propose for Ouroboros, a switch to Ouroboros Praos or Genesis would also bring several advantages. First off, both constructions would give us security in the semi-synchronous model with fully adaptive corruptions (as shown in [DGKR18, BGK+18]), and the use of Ouroboros Genesis would allow newly joining players to bootstrap from the mainchain genesis block only—without the need for a trusted checkpoint—as discussed extensively in [BGK+18].

6.8.2 Snow White

The high-level structure of Snow White execution is similar to the protocols we have already discussed: it contains epochs, committees that are sampled for each epoch based on the stake distribution recorded in the blockchain prior to that epoch, and randomness used for this sampling produced by hashing special nonce values included in previous blocks. Hence, our construction can be adapted to work with Snow White-based blockchains in a straightforward manner.

6.8.3 Algorand

Algorand does not aim for the so-called eventual consensus. Instead it runs a full Byzantine Agreement protocol for each block before moving to the next block, hence blocks are immediately finalized. Consider a setting with $\mathbf{MC}$ and $\mathbf{SC}$ both running Algorand. The main difficulty to address when constructing pegged ledgers is the continuous authentication of the sidechain certificate constructed by $\mathbf{SC}$-maintainers for $\mathbf{MC}$ (other aspects, such as deposits from $\mathbf{MC}$ to $\mathbf{SC}$ work analogously to what we described above). As Algorand does not have epochs, and creating and processing a sidechain certificate for each block is overly demanding, a natural choice is to introduce a parameter R and execute this process only once every R blocks. Namely, every R blocks, the $\mathbf{SC}$-maintainers produce a certificate that the $\mathbf{MC}$-maintainers insert into the mainchain. This certificate most importantly contains:

1. a Merkle commitment to the list of withdrawals in the most recent R-block period;
2. a Merkle commitment to the full, most recent stake distribution $\overline{\mathsf{SD}}_j$ on $\mathbf{SC}$;
3. a sufficient number of signatures from a separate committee certifying the above information, together with proofs justifying the membership of the signature's creators in the committee.

This additional committee is sampled from $\overline{\mathsf{SD}}_{j-1}$ (the stake distribution committed to in the previous sidechain certificate) via Algorand's private sortition mechanism such that the expected size of the committee is large enough to ensure honest supermajority (required for Algorand's security) translates into a strong honest majority within the committee. Note that the sortition mechanism also allows for a succinct proof of membership in the committee. The members of the committee then insert their individual signatures (signing the first two items in the certificate above) into the $\mathbf{SC}$ blockchain during the period of R blocks preceding the construction of the certificate. All the remaining mechanics of the pegged ledgers are a direct analogy of our construction above.

Algorithm 6 The **SC** transaction verifier.

```
1:  function VERIFYTX_SC(t⃗x)
2:      bal[MC] ← Initial MC stake
3:      bal[SC] ← Initial SC stake
4:      mc_outgoing_tx ← ∅; seen ← ∅
5:      for tx ∈ t⃗x do
6:          (txid, lid, (send, sAcc), (rec, rAcc), v, σ, t) ← tx
7:          m ← (txid, lid, (send, sAcc), (rec, rAcc), v)
8:          if ¬Ver(sAcc, m, σ) ∨ seen[txid] ≠ 0 then
9:              continue
10:         end if
11:         if lid = send then
12:             if bal[send][sAcc] − v < 0 then
13:                 continue
14:             end if
15:             if lid = MC ∧ send ≠ rec then
16:                 mc_outgoing_tx[txid] ← t + 2k
17:             end if
18:         end if
19:         if lid = rec then
20:             if send ≠ rec then
21:                 ▷ Effect pre-image tx immature
22:                 if t < mc_outgoing_tx[txid] then
23:                     continue
24:                 end if
25:             end if
26:             bal[rec][rAcc] += v
27:         end if
28:         if lid = send then
29:             bal[send][sAcc] −= v
30:         end if
31:         seen ← seen || tx
32:     end for
33:     return seen
34: end function
```

Algorithm 7 The **SC** verifier.

```
1: function VERIFIER_SC(C_sc, C_mc)
2:     t⃗x ← ANNOTATETX_SC(C_sc, C_mc)
3:     return t⃗x ≠ VERIFYTX_SC(t⃗x)
4: end function
```

Algorithm 8 The **SC** transaction annotation.

```
1:  function ANNOTATETX_SC(C_sc, C_mc)
2:      t⃗x ← ∅
3:      for each time slot t do
4:          t⃗x' ← ϵ
5:          if C_sc has a block generated at slot t then
6:              B ← the block in C_sc generated at t
7:              for tx ∈ B do
8:                  t⃗x' ← t⃗x' || tx
9:              end for
10:         end if
11:         if C_mc has a block generated at slot t then
12:             B ← the block in C_mc generated at t
13:             for tx ∈ B do
14:                 t⃗x' ← t⃗x' || tx
15:             end for
16:         end if
17:         for tx ∈ t⃗x' do
18:             ▷ Mark the time of each tx in t⃗x'
19:             tx.t ← t
20:         end for
21:         t⃗x ← t⃗x || t⃗x'
22:     end for
23:     return t⃗x
24: end function
```

Chapter 7

Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake

7.1 Introduction

A significant limitation of traditional blockchain protocols, such as the Bitcoin, is the fact that the transaction ledger is a public resource and thus information about the way the transaction issuers operate may be leaked to an adversary. This consideration was acknowledged early on and Bitcoin itself [Nak08] includes a number of measures to mitigate transaction privacy loss. Namely users produce a new pseudonymous address for each payment received and addresses from the same wallet can be selected to be indistinguishable from addresses from different wallets. Still, the information available in the blockchain itself is susceptible to analysis and it has been demonstrated early on that significant information can be extracted by clustering the Bitcoin transaction "graph", see e.g., [RS13, MPJ+16].

This state of affairs motivated the development of privacy enhancing and privacy-preserving techniques for distributed ledgers. First, methods such as CoinJoin [Max13] and CoinShuffle [RMK14] were proposed as mechanisms to reduce the effectiveness of de-anonymization techniques based on tracing and clustering. Subsequently redesigned cryptocurrencies were put forth that attempted to introduce stronger privacy-enhancing techniques by design in the distributed ledger protocol. These included Zerocash [BCG+14] and Monero which is based on Cryptonote [vS13]. We note that despite their enhanced privacy characteristics, some leakage still exists in these protocols (even if we exclude leakage on the network layer, which is an issue orthogonal to what these protocols study including the present work). This can be exploited as demonstrated in recent works [KFTS17, MSH+18, KYMM18]. The above privacy-enhancing techniques primarily focused on the transaction processing layer of the distributed ledger leaving the consensus back-end mechanism largely the same.

Concurrently with these developments however, another line of research works in blockchain design focused on resolving fundamental issues with the energy consumption requirements of the underlying proof-of-work (PoW) mechanism of Bitcoin. In particular, this lead to a sequence of works in proof-of-stake (PoS) blockchain protocols that include Algorand [Mic16], Ouroboros [KRDO17], Ouroboros Praos [DGKR18], Ouroboros Genesis [BGK+18], Sleepy-Consensus [PS17], and Snow White [BPS16]. PoS blockchain protocols alleviate the requirement to perform proof-of-work by solving computationally hard puzzles. Instead, they refer to the stake that each participant possesses as reported in the blockchain and, through cryptographic means, elect the next participant to extend the transaction ledger (who is commonly referred as the next *leader* or even *slot leader* when the execution time is divided in slots.) PoS protocols have been touted as the next important advance in real-world distributed ledger systems and a number of well-known cryptocurrencies are in the process of incorporating them into their deployed systems including Ethereum with the Casper protocol [Zam17] and Cardano with Ouroboros [Com18].

The above state of affairs raises an important open question: is it possible to build a PoS-based privacy enhanced distributed ledger? This is the main motivation of this work where we tackle this problem and answer

the question in the affirmative.

Our results. We propose a new formal model for a PoS-based privacy-preserving distributed ledger in the universal composition (UC) setting, [Can01], and a new protocol that realizes it, Ouroboros Crypsinous.[1] Our protocol analysis with respect to the basic properties of consistency and liveness is inspired by Ouroboros Genesis, [BGK+18], a recent (non-private) PoS blockchain protocol formally analyzed in the UC setting. Our protocol provides the first formally analysed PoS-based privacy-enhanced blockchain protocol. Moreover, for the first time our protocol achieves simulation-based privacy that is even universally composable, as well as forward-secure, i.e., it ensures that privacy (as well as consistency and liveness) are preserved independently of any other protocols running concurrently with our ledger implementation, and even under adaptive corruption.

It is worth noting that PoS and transaction privacy is, seemingly, a contradiction in terms: issuing a block by proof-of-stake fundamentally leaks information about the issuer and the state of the ledger. We circumvent the contradiction by designing a new privacy-enhancing PoS operation that, roughly speaking, extends the SNARK machinery of "transaction pouring" in Zerocash to a setting where coins evolve without loosing their value, enabling on the way a proof of stake-eligibility that does not leak any additional information.

The design has several subtleties since a critical consideration in the PoS setting is tolerating adaptive corruptions: this ensures that even if the adversary can corrupt parties in the course of the protocol execution in an adaptive manner, it does not gain any non-negligible advantage by e.g., re-issuing past PoS blocks. In non-private PoS protocols such as Algorand [Mic16] and Ouroboros Genesis [BGK+18] this is captured by employing forward secure signatures. In the context of our protocol however, a more sophisticated combination of key-private forward-secure encryption—a new encryption primitive which we formally define and realize—and an evolving coins mechanism is required to achieve the same level of security. Intuitively, the reason is that we need to ensure that past coins received provide no significant advantage to the adversary when it corrupts an active stakeholder. We note that the naïve approach of simply paying oneself with a new coin does not work here, as the same coin should be able to be elected multiple times in a sequence of PoS invocations without leaving any evidence in the ledger.

Our private ledger formalization is also of independent interest since it captures for the first time the concept of a privacy enhanced transaction ledger in the UC-setting which is generally applicable to both the PoW and PoS settings. Interestingly, we observe that the latter case requires a slightly expanded adversarial interface that allows a sampling of the stakeholder distribution per unit of time (referred to as "slot"). (A similar sampling can be also observed in Bitcoin, but since *miner privacy* is not considered a prime requirement this was never formalized.) Adversarial sampling captures the fact that in the PoS setting traffic analysis is possible based merely on the frequency one entity issues a PoS block. Our formal model ensures that this is the only privacy leakage that will be incurred during the execution of the protocol. A secondary formalization contribution is the concept of UC key-private forward-secure encryption which, even though the two relevant properties were studied independently, a UC functionality capturing both has never appeared until our work.

We note that our work is concurrent, and independent, of another paper on privacy-preserving proof-of-stake by Ganesh et al. [GOT18]. This work focuses on constructing a generic, privacy-preserving leadership election, given a list of commitments to each party's stake. Our work by contrast focuses on ensuring the proof of stake leadership election can run with a provably secure, privacy-preserving transaction scheme. Notably, Zerocash cannot immediately be used with the system of [GOT18], as it does not maintain a list of stake commitments – indeed, such a list would appear to reveal more about the shift in funds than Zerocash does, such as how long an account has seen no changes.

7.2 Protocol Intuition

To begin with, we give a high-level sketch of the Ouroboros Crypsinous protocol in this Section, to aid in understanding the more formal break-down of the protocol in Section 7.6, and to introduce core concepts. We

[1] The word "Crypsinous" is Greek and refers to a person who is mindful of their privacy. We thank Konstantinos Mitropoulos for suggesting it to us.

will first sketch the design of two protocols we are building on – Ouroboros Genesis [BGK+18], and Zerocash [BSCG+14]. We will discuss how these can be combined, and the issues that arise through this combination. Finally, we will sketch how we have resolved these issues.

7.2.1 The Foundations of Genesis and Zerocash

Ouroboros Genesis [BGK+18], divides time into discrete *slots*. At protocol start, parties are assigned initial *stake* in the system. Typically, only the relative amount of such stake is considered, i.e. how much each party holds out of the total stake. By protocol-external means, the distribution of this stake may shift over time, e.g. by users trading it amongst each other. Each slot, users have a probability proportional[2] to their relative stake to be "elected" as a *leader* of the slot. In practice, this relies on a pseudo-random value being below a user-specific target. Such leaders may then create a new block, and sign it with a proof of leadership eligibility. In order to prevent so-called "grinding attacks", in which parties attempt the leadership election arbitrarily often with different accounts, transferring themselves the funds, Genesis divides time further into *epochs*. In each epoch, the distribution of stake considered for leadership is fixed, and the pseudo-random values used to determine it can only be predicted once the epoch starts.

Zerocash [BSCG+14] achieves complete transactional privacy in a distributed ledger setting, through the use of non-interactive zero-knowledge (NIZK) proofs. It represents monetary value through *coins*, which can be created, and spent once. Crucially, it prevents double-spends, and ensures value is preserved, while at the same time preventing the creation and spending of a coin from being linked. A transfer allows spending two coins, and creating two new coins of the same combined value. This closely mirrors the simplest form of Bitcoin transactions. Each party holds a secret key used to spend coins. This secret key is simply a random string, and its corresponding public key is a hash of the secret key. When creating a new coin, it is created *for a public key*. Specifically, a nonce is randomly selected for the new coin, and the transaction creating it commits to the coins public key, nonce, and value. All such created commitments are kept in a protocol-wide Merkle tree. To spend a coin, a party makes a zero-knowledge proof of two things: First, the protocol-wide Merkle tree contains a commitment to it, and second, the spender knows the preimage of the public key. This by itself would allow double spends, so Zerocash reveals a coins *serial number*, which is defined as a PRF of the secret key and the coin's nonce. The transfer finally proves in zero-knowledge that the transaction is zero-sum.

7.2.2 The Core Protocol

The core principle of Ouroboros Crypsinous is combining the strengths of both the Ouroboros Genesis and Zerocash protocols. In particular, we note that while Ouroboros Genesis assumes the distribution of stake to be public, this fact is only used in verifying that leaders of a slot met the appropriate target. To remove this intrinsic leakage, we have parties hold Zerocash-style coins, with each coin being separately considered for leadership. As in Ouroboros Genesis, each coin is eligible to be a leader if a pseudorandom value meets some target. Instead of revealing the coins value, however, in Crypsinous parties produce a NIZK proof of this, as well as proving that the respective coin is unspent. This also forces us to explicitly model the transaction system by which stake is allowed to shift – as the stake distribution is no longer simply supplied to every party by the environment, it is necessary to make explicit how it is derived. For this reason, the core Crypsinous protocol includes a Zerocash-like transaction system.

7.2.3 Freezing Stake in Zero Knowledge

The security argument of Ouroboros Genesis relies on parties not being able to manipulate whether or not they won a leadership election. Specifically, it assumes the distribution of stakeholders to be fixed *before* the randomness for the same epoch is decided. Likewise, the set of coins that are eligible for a slot in the leadership election is fixed in Ouroboros Crypsinous. The protocol maintains this frozen set of coins, $\mathfrak{C}^{\mathsf{lead}}$, separately to the set

[2] We note that although it is not technically linear, this is a close approximation.

of coins usable for spending, $\mathfrak{C}^{\mathsf{spend}}$. In practice, as coins are anonymously as sets of commitments and serial numbers, and as any reuse of a serial number would lead to some privacy leakage, we represent them through two sets of commitments, $\mathbb{C}^{\mathsf{lead}}$ and $\mathbb{C}^{\mathsf{spend}}$, and one set of serial numbers, $\mathbb{S}$. In creating the leadership proofs, a coins serial number is revealed. As it may later be spent, this would lead to some privacy leakage. To mitigate this, we instead *evolve* the coin in the leadership transaction. This new, evolved coin can then be spent, and used in further leadership proofs, the latter being possible as it is derived deterministically from the former coin, which does not allow influencing the probability of it being elected in the remainder of the epoch. We note that as this design inherently destroys the old coin, it is important that even leadership transactions of different branches of the chain are imported and validated.

7.2.4 Adaptive Corruptions

As Ouroboros Genesis is secure in the adaptive corruption model, it seems natural that privacy results should be possible in the same model. The construction described so far, is not directly secure against adaptive corruptions. An adversary could, after corrupting a party, attempt to create leadership proofs of past slots with the newly corrupted party. Further, we note that – in the UC framework – a non-committing encryption would be needed for the ciphertexts in the Zerocash style transactions, as with a committing encryption, the simulator would be unable to produce ciphertexts that stand up to inspection after corruption.

We solve the former issue, by adding a cheap key-erasure scheme into the NIZK for leadership proofs. Specifically, parties have a Merkle tree of secret keys, the root of which is hashed to create the corresponding public key. The Merkle tree roots acts like a Zerocash coin secret key, and can be used to spend coins. For leadership however, parties also must prove knowledge of a path in the Merkle tree to a leaf at the index of the slot they are claiming to lead. After a slot passes, honest parties erase their preimages of this part of that path in the tree. As the size of this tree is linear with the number of slots, we allow parties to keep it small, by restricting its size. Keys therefore are associated with their creating time, by committing to this in the corresponding public key. While this does mean keys can expire, we note parties can trivially refresh them, and further will sketch in Section 7.8 that this is a rare occurrence for practical parameters. We emphasize that parties *are* able to spend and refresh keys, even when expired.

While we could easily present Ouroboros Crypsinous using non-committing encryption, known realizations of this primitive are not efficient enough for this purpose in practice. Instead, we take advantage of our protocols network assumptions, which include an upper bound on message delivery, $\Delta_{\max}$. This allows us to utilize forward secure encryption instead of non-committing encryption, under the assumption that corruption is "delayed" by $\Delta_{\max}$. This delay is modeled by restricting adversarial access to the forward secure encryption secret key at time τ to the key for time $\tau + \Delta_{\max}$.

7.3 The Model

Following the recent line of works proving composable security of blockchain ledgers [BMTZ17, BGK+18] we provide our protocol and security proof in Canetti's universal composition (UC) framework [Can01]. In this section we discuss the main components of the real-world execution, including the hybrid functionalities that the protocol uses. We discuss the ideal world, and in particular the private transaction ledger functionality in Section 7.5. We assume that the reader is familiar with simulation-based security and has basic knowledge of the UC framework. We provide all the aspects of the execution model from [BMTZ17, BGK+18] that are needed for our protocol and proof, but omit some of the low-level details and refer the more interested reader to these works wherever appropriate. We note that for obtaining a better abstraction of reality, some of our hybrids are described as global (GUC) setups [CDPW07]. The main difference of such setups from standard UC functionalities is that the former are accessible by arbitrary protocols and, therefore, allow the protocols to share their (the setups') state. The low-level details of the GUC framework—and the extra points which differentiate it from UC—are not necessary for understanding our protocols and proofs; we refer the interested reader to [CDPW07] for these

details. We will use sid as a session identifier throughout the paper.

Protocol participants are represented as parties—formally Interactive Turing Machine instances (ITIs)—in a multi-party computation. We assume a central adversary $\mathcal{A}$ who corrupts stakeholders and uses them to attack the protocol. The adversary is *adaptive*, i.e., can corrupt additional stakeholders at any point and depending on his current view of the protocol execution. We cast our protocols in the partially synchronous communication version of UC proposed in [BMTZ17]: parties have access to a global clock setup, denoted by $\mathcal{G}_{\text{CLOCK}}$, and can communicate over a network of authenticated multicast channels with a bounded delay Δ denoted by $\mathcal{F}^{\Delta}_{\text{N-MC}}$. Every honest party can send a message thought $\mathcal{F}^{\Delta}_{\text{N-MC}}$ to all other honest parties but the adversary can delay its delivery to any honest party by a number of rounds of his choice but no greater than Δ. Honest receivers cannot tell when a message will arrive as they know neither when the message was sent nor the delay Δ. As in the case of Bitcoin, cf. [GKL17, PSS17, BGK+18], our protocol is implicitly aware of an overestimate Δ_{max} of the actual (unknown) network delay Δ. However, this Δ_{max} is not used in the message passing; instead the protocol proceeds in an optimistic manner once messages are received (after at most Δ rounds from sending) and Δ_{max} is only used in the staking procedure to determine the leader(s) of each slot.

Similarly to [BMTZ17, BGK+18], for UC realization in such a globally synchronized setting, the target ideal functionality, i.e., the ledger, needs to keep track of the number of activations that an honest party gets—so that it can enforce in the ideal world the same pace of the clock as in the real world. This is achieved by describing the protocol so that it has an (implicit) predictable behavior of clock interactions for any given activation pattern—which the ideal functionality can (and will) mimic. We refer to [BMTZ17] for details.

We adopt the *dynamic availability* model implicit in [BMTZ17] which was fleshed out in [BGK+18]. We next sketch its main components: All functionalities, protocols, and global setups have a dynamic party set, i.e., they all include special instructions allowing parties to register, deregister, and allowing the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and deregister with them, and they also allow other setups to learn their set of registered parties.

Utilizing the full dynamic availability model results in separating the honest parties in the following categories: *offline* parties are honest parties that are deregistered from the network functionality. Parties which are not offline are separated into two (sub-)categories, called *(fully) online*—parties which are registered with all their setups and ideal resources—and *(online but) stalled*—parties that are registered with their local network functionality, but are unregistered with at least one of the global setups. Each of these (non-offline) subclasses is further split into two subcategories along the lines of the classification of [BMTZ17]: those that have been in a non-offline state for more that `Delay` rounds—where `Delay` is a ledger parameter—are *synchronized*, whereas the remainder are *de-synchronized*. Our protocol makes use of the following hybrid functionalities from [BGK+18]. (The ideal world execution makes access to the global setups presented below and the private ledger functionality which is presented in Section 7.5.)

- The global clock functionality $\mathcal{G}_{\text{CLOCK}}$ which keeps track of the current (global) round and reports it to any party that requests it. The round advances whenever all honest (currently registered) parties and functionalities inform $\mathcal{G}_{\text{CLOCK}}$ that they are finished with their current round's actions (note that this is not a communication round).
- The bounded-delay authenticated channels network $\mathcal{F}^{\Delta}_{\text{N-MC}}$ described above.
- The genesis block generation and distribution functionality $\mathcal{F}_{\text{INIT}}$, which captures the assumption that all parties (old and new) agree on the first, so-called *genesis* block. In fact, this functionality is slightly different from the one in [BGK+18] as the blocks in our work have a different structure to ensure privacy. Concretely, In Ouroboros-Genesis this block includes the keys, signatures, and original stake distribution of the parties that are around at the beginning of the protocol. Here, for each stakeholder registered at the beginning of the protocol, $\mathcal{F}_{\text{INIT}}$ records his keys and initial coin commitments in the genesis block; this block is distributed to anyone who requests it in any future round. As in [BGK+18] we assume wlog that the global time is $\tau = 0$ in the genesis round. We refer to Section 7.9 for a description of our new genesis block functionality.
- A global random oracle $\mathcal{G}_{\text{RO}}$ for abstracting hash function queries. As typically in cryptographic proofs the queries to hash function are modeled by assuming access to a random oracle: Upon receiving a query

$(\text{EVAL}, \text{sid}, x)$ from a registered party, if x has not been queried before, a value y is chosen uniformly at random from $\{0,1\}^\kappa$ (for security parameter κ) and returned to the party (and the mapping (x, y) is internally stored). If x has been queried before, the corresponding y is returned. As in [BGK+18] we capture this by a global random oracle (GRO), i.e., a global setup that behaves as above.

To ensure privacy of transactions, we need to equip our model with a couple of extra functionalities not present in previous works. For instance, the (non-private) Ouroboros protocol-line [DGKR18, BGK+18] relies on verifiable random functions and key-evolving signatures to ensure security of the lottery which defines slot leaders and prevent double spending in the presence of an adaptive adversary.

In our protocol we cannot use signatures to authenticate coins/transactions as we need to keep the spent amount and the identities of the receiver private. For this reason we introduce *key-private forward secure encryption* and non-interactive zero-knowledge proofs (NIZKs). Our protocol will be described as having access to hybrid-functionalities for these primitives. These functionalities along with their implementation from a common reference string (CRS) and their security proofs are described in Section 7.4. To our knowledge no definition of key-private forward secure encryption or an implementation thereof has been suggested. In fact, for reasons discussed below (see Section 7.4.2) an implementation of this primitive against fully adaptive adversaries might be impossible without additional setup assumptions. Instead, here we make an assumption about the (in)ability of the adversary to quickly read keys of newly corrupted parties and prove the security of our protocols under this assumption. Proving impossibility of the primitive against a fully adaptive adversary (or providing a protocol for it) is an interesting future direction.

Finally, our construction will make use of non-interactive equivocal commitments and pseudo-random functions (PRFs). Construction of both these primitives exists assuming a CRS under standard hardness assumption, e.g., hardness of the DDH (Decision Diffie Hellman) problem.

Remark 1: (Assumptions on the environment/adversary as functionality wrappers.) The security statements about implementation of ledgers are typically conditional. E.g., the Bitcoin ledger is proved secure assuming the majority of the system's hashing power is honest, and the Ouroboros (Genesis) ledger is implemented assuming the majority of the stake is held by honest parties. These assumptions can be easily described by explicitly restricting the class of environments and adversaries, but this would sacrifice the universal composability of the statement. We follow the paradigm of [BMTZ17] to capture these assumptions without compromizing composability: Instead of explicitly restricting the adversary and environment, we introduce a functionality wrapper that wraps the functionalities that the protocol accesses and forces the required assumptions on the adversary/environment. We refer to [BMTZ17] for a more detailed discussion. As a forward pointer, the wrapper used in our security statements is sketched in Section 7.13. As this wrapper only becomes relevant for interpreting our main theorems (Theorem 17 and 18) it might be easier for the first-time reader to postpone parsing it until then.

7.4 Tools

In this section we describe the main tools used by Ouroboros Crypsinous: non-interactive zero-knowledge proofs (NIZKs), key-private forward secure encryption, maliciously-unpredictable PRFs (MUPRFs), and equivocal commitments. We describe ideal functionalities capturing NIZK and key-private forward-secure encryption, and refer to their UC implementations in Sec 7.14 and 7.15. Ouroboros Crypsinous is described and proved secure assuming hybrid access to these ideal functionalities and its security when these functionalities are replaced by their implementations will follow directly from the universal composition theorem.

Further, we define the properties satisfied by MUPRFs and equivocal commitments.

7.4.1 Non-Interactive Zero Knowledge

We utilize the Non-Interactive Zero Knowledge functionality $\mathcal{F}_{\text{NIZK}}$ and protocol of [KZM+15b] , (for completeness, $\mathcal{F}_{\text{NIZK}}$ is described in detail in Section 7.9). This functionality allows generating proofs π that a statement

$\mathbf{x}$ is in a given NP language $\mathcal{L}$, with a witness $\mathbf{w}$. We use the "weak" functionality suggested, which permits an adversary to generate new proofs for already proven statements.

We note that while [KZM⁺15b] provides a construction, it is only shown to satisfy game-based properties. We formally prove its security in the UC setting assuming a CRS in Section 7.14.

NIZKs can be used for signature-like behaviour by embedding the messages that are to be signed in the statements of simulation-extractable NIZKs, constructing in this way a *signature of knowledge* [GM17] (SoK). In particular, we note that witnesses used to generate proofs in Ouroboros Crypsinous will contain the party's secret key, and the proved statement commits to the party's public key. As a result, the NIZK used in Ouroboros Crypsinous has similar unforgeability properties as standard signatures.

7.4.2 Key-private Forward-Secure Encryption

To guarantee the forward-privacy of transactions, a forward-secure encryption scheme [CHK03] is necessary to hide information sent encrypted to a party's long-term encryption secret key. Note that traditional forward-secure encryption is insufficient, as it would leak information about the recipient of a transaction. To preserve the recipient's anonymity in Crypsinous transactions, we therefore require key-privacy as well [BBDP01]. Furthermore, as the simulator must create simulated ciphertexts, which it may later need to reveal the message of, encryption in the UC setting needs to be non-committing to withstand adaptive corruptions. Interestingly, however, there are no existing encryption schemes that simultaneously achieve key-privacy, forward-security, and the non-commitment property.

We overcome the above limitation by introducing a slightly weakening the above security requirements and only requiring forward-security with a time-sensitive non-committing property: Informally, only messages addressed to a time window of size Δ_{max} into the future are protected. As it turns out, this weaker notion is sufficient for our purposes. Even for this notion, however, it is not evident how to efficiently realise such an encryption in the UC setting. To understand the issue, it is useful to recall how we can realize non-interactive non-committing encryption via erasures. The idea is to have parties update their keys once the message is received. More concretely, a message is encrypted at round τ and sent over to the receiver so that it can be decrypted with key sk_{τ}^{ENC}. Upon receiving it, the receiver can decrypt it (using sk_{τ}^{ENC}), and immediately update the key to $sk_{\tau'}^{\text{ENC}}$ for the next round (and erase sk_{τ}^{ENC}). This way the link between the ciphertext and the key is eliminated by the time the adversary corrupts the receiver.

The above approach clearly fails if the channel has any delay, as in out setting, as this gives the adversary a window of opportunity of size Δ, and bounded only by Δ_{max}, to attack during which the message is already being transmitted but has not yet been received by the recipient. This makes erasures useless in this window (if correctness is to be maintained).

To bypass the above issue, we make an assumption on the adversary's adaptiveness which, roughly, implies that the adversary cannot immediately access the secret key of a newly corrupted party. Specifically, we assume that the adversary corrupting a party with key sk_{τ}^{ENC} at time τ does not receive sk_{τ}^{ENC}, but rather the key $sk_{\tau+\Delta_{\text{max}}}^{\text{ENC}}$, which this party would hold in time $\tau+\Delta_{\text{max}}$, if it were allowed to properly update its key. We emphasize that this is a milder assumption than that of delayed party-corruption which underlines the security of [KRDO17, BPS16]. Indeed, in these works the adversary is forbidden from accessing the entire state of a corrupted party for a certain number of rounds after corruption; instead, here we only restrict his access to the present keys, and we even give the adversary an outlook, already upon corruption, of how the key will look in the near future.

To enforce the above restriction without affecting the universal composability of our statements, we use a technical trick inspired by [BMTZ17, DGHM13] (related to the wrappers used in Remark 1.): Concretely, we introduce an ideal functionality which captures this restriction/assumption. This functionality, denoted by $\mathcal{F}_{\text{KEYMEM}}$, stores keys upon request from parties, and updates them every round using a one-way function Update; when an honest party requests a key it has submitted in the past, the functionality sends it the current key. However, when the adversary asks for a key (on behalf of a corrupted party) $\mathcal{F}_{\text{KEYMEM}}$ first applies Update Δ_{max} times, and returns the updated key to the adversary.

Note that the direct way of enforcing the assumption would be to limit all our statements to only apply to a restricted class of adversaries. For reasons similar to the discussion in Remark 1 above, this would immediately imply that universal composition no longer holds.[3] As an added bonus from using the above functionality-based approach for restricting the adversary, our treatment ensures that the restriction is localized to the encryption functionality; thus, if someone comes up with an instantiation of the encryption functionality against a fully adaptive adversary, or protocol would immediately be secure against such an adversary. The $\mathcal{F}_{\text{KEYMEM}}$ functionality is specified below.

Functionality $\mathcal{F}_{\text{KEYMEM}}$

$\mathcal{F}_{\text{KEYMEM}}$ is parameterized by its corruption delay Δ_{max}, and a memory update function Update. It maintains a memory M_p for each party U_p, initialized to ϵ, as well as a flag isInit_p for each party U_p, initialized to $\perp$. We write $\mathsf{Update}^{\Delta}_{\text{max}}$ to mean "apply Update Δ_{max} times."

On receiving a message (Init, sid, M') *from* U_p*:* If $\mathsf{isInit}_p = \perp$, let $M_p \leftarrow M'$; $\mathsf{isInit}_p \leftarrow \top$.

On receiving a message (Get, sid) *from* U_p*:* If $\mathsf{isInit}_p = \top$, return M_p if U_p is honest, otherwise return $\mathsf{Update}^{\Delta}_{\text{max}}(M_p)$.

On receiving a message (Update, sid) *from* U_p*:* If $\mathsf{isInit}_p = \top$, update $M_p \leftarrow \mathsf{Update}(M_p)$.

The UC functionality for key-private and forward-secure encryption, $\mathcal{F}_{\text{FwEnc}}$, is described in detail in Appendix 7.9, and the accompanying construction is described below.

We extend the notion of forward-secure encryption (FSE) with a notion of *key privacy*, described in detail in Definition 21 below. While this definition itself is novel, it is possible to combine existing schemes to satisfy it. In particular, [CHK03] constructs FSE from hierarchical identity-based encryption (HIBE). Their scheme, paired with the anonymous HIBE construction of [BW06] satisfies our requirements of key-privacy as we will argue below.

For the argument of key privacy, the FSE from HIBE construction in [CHK03] is straightforward, with the ciphertexts simply being the underlying HIBE scheme's ciphertexts. The core argument of the anonymity of [BW06], is the indistinguishability of ciphertexts from random group elements – and therefore their independence of the encrypting identity (cf. [BW06, Lemmas 8& 9]. We note that the ciphertexts' pseudo-randomness implies a stronger notion than just anonymity – the ciphertext also does not reveal any information about the HIBE public key. In particular, as ciphertexts are indistinguishable, our enhanced security game given in Definition 21 is satisfied. The game, as well as the subsequent UC construction, can be found in Section 7.15.

This construction's time and space complexity is logarithmic to the number of time slots. As the number of slots is by necessity less than 2^{κ}, the use of this forward-secure encryption has a linear increase in cost with respect to the security parameter compared to standard encryption.

Key-Private Forward-Security Against Chosen Ciphertext Attacks

Definition 21. A key-evolving public-key encryption scheme is *key-privately forward-secure against chosen ciphertext attacks (kp-fs-CCA)* if any PPT adversary has only negligible advantage $|2 \cdot \Pr[b' = b] - 1|$ in the following game:
Setup: For each party $U_p \in \mathcal{P}$, run $(pk_p, sk_p^0) \xleftarrow{\$} \mathsf{Gen}(1^{\kappa}, N)$. The adversary receives all public keys pk_p. Further, a bit $b \leftarrow \{0,1\}$ is selected, but not revealed to the adversary.
Attack: The adversary issues multiple $\mathsf{challenge}(j, (U_0, m_0), (U_1, m_1))$ queries, multiple $\mathsf{corrupt}(i, U_p)$ queries, and multiple $\mathsf{decrypt}(k, c, U_p)$ queries, where $U_p, U_0, U_1 \in \mathcal{P}$, and $0 \le i \le N; 0 \le j \le N; k \le N$. Further,

[3] One could attempt to prove a tailored, weaker, and non-universal composition theorem that would apply only to the restricted class of adversaries considered in our encryption-scheme's security proof. But this is not in the spirit of our treatment which explicitly aims at fully-composable (UC) protocols, and it is also rendered unnecessary using our trick above.

if a corrupt query is made for some party a challenge query is also made for, then the corresponding i must be greater than the corresponding j. corrupt queries may be issued only once for each party.

- $\mathsf{corrupt}(i, U_p)$ is answered with $sk_p^i \triangleq \mathsf{Upd}(\ldots \mathsf{Upd}(sk_p^0, 1), \ldots, i)$.
- $\mathsf{challenge}(j, (U_0, m_0), (U_1, m_1))$ is answered by responding with $c = \mathsf{Enc}_{pk_{U_b}}(j, m_b)$, and (j, c, U_0) and (j, c, U_1) are recorded as challenges.
- $\mathsf{decrypt}(k, c, U_p)$ is answered with $\perp$ if (k, c, U_p) is recorded as a challenge. Otherwise, it is answered with $\mathsf{Dec}_{sk_p^k}(k, c^*)$.

Guess: The adversary outputs a guess $b' \in \{0, 1\}$, and wins the game iff $b' = b$.

7.4.3 PRFs with unpredictability under malicious keys

Consider a PRF family $\{f_k\}_{k \in K}$ such that $f_k : X \to Y$ for all $k \in K$. The usual PRF security requires that any PPT distinguisher $\mathcal{D}$ with an oracle cannot tell the difference between an oracle $f_k(\cdot)$, for a randomly selected k and a truly random function over $X \to Y$. The definition can be ported to the random oracle setting where both the function f_k as well as the distinguisher D have access to a random oracle $H(\cdot)$. Unpredictability under malicious key generation, is an additional property that, intuitively, suggests the function does not have any "bad keys" that can eliminate the entropy of the input, a concept introduced in [DGKR18]. In the random oracle model, the property can be expressed as follows: for any PPT $\mathcal{A}$ and $x \in X, T \in \mathbb{N}$, the probability of the event $\Pr[f_k(x) < T | x \notin Q_H]$ equals $T/2^\kappa$ where $\mathcal{A}(1^\kappa) = k$, and Q_H is the set of queries of $\mathcal{A}$ to H.

We will employ the following construction. Let $H : \{0,1\}^* \to \langle g \rangle$ be a function mapping to a cyclic group generated by g with a compact representation. We use an elliptic curve group based on the "elligator" curves [BHKL13] that have the property that a uniform element over $\langle g \rangle$ is indistinguishable from a random κ-bit string. We then define $f_k(m) \mapsto H(m)^k$ for $k \neq 0$ and we show that it is a PRF with unpredictability under malicious key generation from X to $\{0,1\}^\kappa$. Indeed observe first that $\langle g^k, H(m), H(m)^k \rangle$ is a DDH triple over the group $\langle g \rangle$. Thus, by the DDH assumption and the random oracle model, we can substitute all queries to the PRF by random group elements. Now observe that by the encoding properties of the curve these elements can be substituted by random strings over $\{0,1\}^\kappa$. Regarding the unpredictability under malicious key generation observe that in the random oracle model, $\Pr[H(x)^k < T] \leq \sum_{y<T} \Pr[H(x)^k = y] = T \cdot \Pr[H(x) = y^{1/k}] \leq T/2^\kappa$ in the conditional space $x \notin Q_H$.

7.4.4 Equivocal Commitments

We make use of a standard non-interactive equivocal commitment scheme, which is secure in the CRS model assuming hardness of discrete logarithms (cf. [DG03]). For self-containment we include a high-level description, including some notation used in our proofs below.

Specifically, we will assume the existence of six algorithms, $\mathsf{Init}_{\mathsf{comm}}$, Comm, DeComm, $\widehat{\mathsf{Init}}_{\mathsf{comm}}$, $\widehat{\mathsf{Comm}}$, and Equiv. $\mathsf{Init}_{\mathsf{comm}}$ generates a public key pk^{COMM} which is given as an argument to Comm and DeComm and will be part of the parameterization of the CRS functionality. In addition to satisfying the traditional commitment properties, of binding, hiding, and correctness, the scheme also satisfies equivocality. Specifically, $\widehat{\mathsf{Init}}_{\mathsf{comm}}$ provides an equivocation key in addition to pk^{COMM}. This equivocation key "breaks" the binding property – $\widehat{\mathsf{Comm}}$ can generate a commitment without a message, and Equiv can later create a witness matching any message for this commitment. We note that we do not require additional common properties, such as extraction or non-malleability, as these are provided by other components of Ouroboros Crypsinous' design, in particular the NIZK functionality.

We write $(cm, r) \leftarrow \mathsf{Comm}(m)$ to create the commitment cm for message m, and $\mathsf{DeComm}(cm, m, r) = \top$ if the decommitment to m and r verifies. Likewise, we write $cm \leftarrow \widehat{\mathsf{Comm}}(ek)$ for simulating a commitment with equivocation key ek, and $r \leftarrow \mathsf{Equiv}(ek, cm, m)$ to equivocate, where $\mathsf{DeComm}(cm, m, r) = \top$. In all these, we leave the public key pk^{COMM} implicit, as it is assumed to be globally known via the CRS.

7.5 The Private Ledger

We next provide the complete description of the private ledger functionality that, as we prove, is implemented by Ouroboros Crypsinous. To describe how privacy is captured in the Crypsinous ledger, we first recall how submitted transactions are stored in the original—non-private—ledger from [BMTZ17, BGK+18]: When a transaction tx is submitted, the ledger creates—and stores in the buffer—an *annotated version* of the transaction tx, denoted as $\mathsf{BTX} := (\mathsf{tx}, \mathrm{txid}, \tau_L, U_s)$, which includes several useful metadata: txid is a unique identifier for this transaction, τ_L is the clock value when the transaction is received, and U_s is the ID of the party that submitted the transaction. Note that this metadata is used for internal bookkeeping and is not necessarily included in the state of the ledger when (and if) the transaction makes it there. In fact, whether or not this data is included in the state is mandated by the Blockify function of [BMTZ17, BGK+18], a parameter to the private ledger. Nonetheless, in the non-private ledger case, the metadata is handed also to the simulator whenever a transaction is given to him.

Privacy of Crypsinous is captured by the following modifications: First, transactions returned from the functionality are *blinded* by a function BlindTx which is a parameter of a functionality. This function hides any information a party should not see from the ledger state, while state validation operates over the entire, non-blinded state. Further, we parameterize the private ledger with a general purpose leakage algorithm, Lkg, which may additionally leak any function of the ledger state to the adversary.

As a technicality, as the BlindTx function must be applied to "blockified" states, however the structure of these is not known in general, an additional "state blinding" algorithm is accepted as a parameter, which we require to behave equivalently to first blinding all transactions, then passing them to Blockify. Intuitively, for any given state state, $\mathsf{Blind}(\mathcal{P}, \mathsf{ids}, \mathsf{state})$ returns state with every transaction tx replaced by $\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, \mathsf{tx})$. In particular, where $\beta \triangleq \mathtt{map}(\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}))$, we require that:

$$\mathsf{Blind}\left(\mathcal{P}, \begin{matrix} \mathsf{Blockify}(\vec{\mathsf{tx}}_1) \parallel \\ \dots \quad \parallel \\ \mathsf{Blockify}(\vec{\mathsf{tx}}_\ell) \end{matrix}\right) = \begin{matrix} \mathsf{Blockify}(\beta(\vec{\mathsf{tx}}_1)) \parallel \\ \dots \quad \parallel \\ \mathsf{Blockify}(\beta(\vec{\mathsf{tx}}_\ell)) \end{matrix}$$

$\mathsf{Blind}_{\mathcal{A}}$ is defined the same as Blind, but with calls to BlindTx replaced with calls to $\mathsf{BlindTx}_{\mathcal{A}}$.

Functionality $\mathcal{G}_{\mathrm{PL}}$

$\mathcal{G}_{\mathrm{PL}}$ is parameterized by seven algorithms, Validate, ExtendPolicy, Blockify, Lkg, BlindTx, Blind, and predict-time, along with three parameters: $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$, and $\mathfrak{C}_1 := \{(U_1, s_1), \dots, (U_n, s_n)\}$. These parameters are all publicly known. The functionality manages variables state, NxtBC, buffer, τ_L, and $\vec{\tau}_{\mathsf{state}}$, as described in [BMTZ17, BGK+18], as well as a sequence of generated IDs, ids. The variables are initialized as follows: $\mathsf{state} := \vec{\tau}_{\mathsf{state}} := \mathsf{NxtBC} := \mathsf{ids} := \varepsilon$, $\mathsf{buffer} := \varnothing$, $\tau_L = 0$.

The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (following the definition of de-synchronized from above). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\varnothing$. When a new honest party is registered at the ledger, if it is registered with the clock and the global RO already, then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is de-registered, it is removed from $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \varnothing$. Finally, during registration, $(\textsc{generate}, \mathrm{sid}, \textsc{coin})$ is run once, and U_p is replaced with the resulting id in $\mathfrak{C}_1$. Further, the registration procedure returns id.
For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer pt_p (initially set to 1) and a current-state view $\mathsf{state}_p := \varepsilon$ (initially set to empty). We refer to the vector $\mathsf{pt}_1, \dots, \mathsf{pt}_n$ as $\vec{\mathsf{pt}}$. The functionality also keeps track of the timed honest-input sequence (cf. [BMTZ17]) in a vector $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

Handling initial stakeholders: If during round $\tau = 0$, the ledger did not received a registration from each initial stakeholder, i.e., $(U_p, s_p) \in \mathfrak{C}_1$, the functionality halts.

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$; upon receiving response $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ set $\tau_L := \tau$ and do the following if $\tau > 0$ (otherwise, ignore input):

1. Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of de-synchronized honest parties that have been registered (continuously) since time $\tau' < \tau_L - \texttt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.

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

   (a) If $I = (\text{SUBMIT}, \text{sid}, \mathsf{tx})$, set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T \,\|\, ((\text{SUBMIT}, \text{sid}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{tx})), U_p, \tau_L)$; else set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T \,\|\, (I, U_p, \tau_L)$

   (b) Compute $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathsf{NxtBC}, \mathsf{buffer}, \vec{\tau}_{\mathsf{state}})$ and if $\vec{N} \neq \varepsilon$ set $\mathsf{state} := \mathsf{state} \,\|\, \mathsf{Blockify}(\vec{N}_1) \,\|\, \ldots \,\|\, \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathsf{state}} := \vec{\tau}_{\mathsf{state}} \,\|\, \tau_L^\ell$, where $\tau_L^\ell := \tau_L \,\|\, \ldots \,\|\, \tau_L$.

   (c) For each $\mathsf{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \vec{\mathsf{pt}}, \mathcal{H}, \mathsf{ids}) = 0$ then delete BTX from buffer. Also, reset $\mathsf{NxtBC} := \varepsilon$.

   (d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathsf{state}| - \mathsf{pt}_j > \texttt{windowSize}$ or $\mathsf{pt}_j < |\mathsf{state}_j|$, then set $\mathsf{pt}_k := |\mathsf{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

3. If the calling party U_p is *stalled* (according to the definition above), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\text{PL}}$ executes the corresponding code from the following list:

   – *Submitting a transaction:*
   If $I = (\text{SUBMIT}, \text{sid}, \mathsf{tx})$ and is received from a party $U_p \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party U_p) do the following

   (a) Choose a unique transaction ID txid and set $\mathsf{BTX} := (\mathsf{tx}, \text{txid}, \tau_L, U_p)$.

   (b) If $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \vec{\mathsf{pt}}, \mathcal{H}, \mathsf{ids}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \{\mathsf{BTX}\}$.

   (c) Send $(\text{SUBMIT}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{BTX}))$ to $\mathcal{A}$.

   – *Generating IDs:*
   If $I = (\text{GENERATE}, \text{sid}, \mathsf{tag})$ is received from a party $U_p \in \mathcal{P}$, query the adversary with $(\text{GENERATE}, \text{sid}, U_p, \mathsf{tag})$, denoting the response id. Ensure the response is unique for tag and not equal to $\perp$, and record $\mathsf{ids} \leftarrow \mathsf{ids} \,\|\, (U_p, \mathsf{tag}, \mathsf{id})$. Return id.

   – *Reading the state:*
   If $I = (\text{READ}, \text{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\mathsf{state}_p := \mathsf{state}|_{\min\{\mathsf{pt}_p, |\mathsf{state}|\}}$ and return $(\text{READ}, \text{sid}, \mathsf{Blind}(\{U_p\}, \mathsf{ids}, \mathsf{state}_p))$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathsf{Blind}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{state}), \mathsf{map}(\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}), \mathsf{ids}, \mathsf{buffer}), \mathsf{Lkg}(\mathsf{state}, \mathsf{buffer}, \tau_L), \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

   – *Maintaining the ledger state:*
   If $I = (\text{MAINTAIN-LEDGER}, \text{sid})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

   – *The adversary proposing the next block:*
   If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \ldots, \text{txid}_\ell))$ is sent from the adversary, update NxtBC as follows:

(a) Set $\text{listOfTxid} \leftarrow \epsilon$

(b) For $i = 1, \dots, \ell$ do: if there exists $\mathsf{BTX} := (x, \text{txid}, \tau_L, U_j) \in \mathsf{buffer}$ with ID $\text{txid} = \text{txid}_i$ then set $\text{listOfTxid} := \text{listOfTxid} \,\|\, \text{txid}_i$.

(c) Finally, set $\mathsf{NxtBC} := \mathsf{NxtBC} \,\|\, (\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.

– *The adversary setting state-slackness:*
If $I = (\text{SET-SLACK}, (U_{i_1}, \widehat{\mathsf{pt}}_{i_1}), \dots, (U_{i_\ell}, \widehat{\mathsf{pt}}_{i_\ell}))$, with $\{U_{p_{i_1}}, \dots, U_{p_{i_\ell}}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:

(a) If for all $j \in [\ell] : |\mathsf{state}| - \widehat{\mathsf{pt}}_{i_j} \leq \mathtt{windowSize}$ and $\widehat{\mathsf{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, set $\mathsf{pt}_{i_1} := \widehat{\mathsf{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$.

(b) Otherwise set $\mathsf{pt}_{i_j} := |\mathsf{state}|$ for all $j \in [\ell]$.

– *The adversary setting the state for desychronized parties:*
If $I = (\text{DESYNC-STATE}, (U_{i_1}, \mathsf{state}'_{i_1}), \dots, (U_{i_\ell}, \mathsf{state}'_{i_\ell}))$, with $\{U_{i_1}, \dots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathsf{state}_{i_j} := \mathsf{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.

7.5.1 Blinding for Forward-Secure Transactions

In order to define blinding on transactions, we first define transactions as consisting of a vector of subtransactions, denoted $\mathsf{tx} \triangleq (\mathsf{stx}_1, \mathsf{stx}_2, \dots, \mathsf{stx}_\ell)$. Each subtransaction consists of a recipient public key pk_r, and an arbitrary message x, that is $\mathsf{stx} \triangleq (pk_r, x)$. In this context, pk_r is either a public key, generated by a party with an GENERATE query, or the special symbol PUBLIC, denoting the subtransaction is publicly readable. We do not leak the entire annotated transaction to the adversary. Instead, the adversary is shown a modified vector tx, with subtransactions addressed to honest parties replaced with $\perp$. While we do not go into the detail of transfer transactions here, we also replace components referring to already spent coins – for honest parties or adversarial – with $\perp$. This guarantees forward privacy of past transactions, as even on corruption, the adversary cannot retrieve this information. Concretely, we define *blinding* functions $\mathsf{BlindSTx}$ and $\mathsf{BlindTx}$, described below, which hide parts of the ledger from read requests.

$\mathsf{BlindTx}$ takes as input the full ledger state state, an annotated transaction $\mathsf{BTX} = (\mathsf{tx}, \text{txid}, \tau_L, U_s)$, a set of parties $\mathcal{P}$, and the set of generated ids ids. It returns a vector consisting only of the components of the transaction that are readable by some party $U_p \in \mathcal{P}$. An adversarial version of $\mathsf{BlindTx}$, $\mathsf{BlindTx}_{\mathcal{A}}$, additionally returns the time of submission, τ_L, and the submitter U_s[4]. Below, we make use of the commonly used higher-order function map, which applies a function to a list element-wise.

function $\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (pk, \mathsf{stx}))$
 $b \leftarrow 0$
 if stx is not a change subtransaction **then**
  $b \leftarrow 1$
 end if
 if stx is a receipt subtransaction of an already spent coin **then**
  $b \leftarrow 1$
 end if
 if $pk \neq$ PUBLIC $\vee\ pk$ not owned by $U_p \in \mathcal{P}$ **then**
  $b \leftarrow 0$
 end if
 if $b \vee \mathsf{stx}$ is a receipt subtransaction for an adversarial coin **then**

  return (pk, stx)

[4] If we assumed an anonymous broadcast, the submitter would not be needed to be leaked, i.e., the requirement of leaking the submitter is strictly due to network leakage.

else

return $(\bot, |\mathsf{stx}|)$
end if
end function

$$\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathrm{txid}, \cdot, \cdot)) \triangleq \mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathrm{txid})$$

$$\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathrm{txid}, \tau_L, U_s)) \triangleq (\mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathrm{txid}, \tau_L, U_s)$$

7.5.2 Leakage for Leader-Based Protocols

In our system, we permit the leakage $\mathsf{Lkg}_{\mathsf{lead}}$, which effectively simulates the protocols leadership election, and leaks the winning party. Specifically, for each time τ, the adversary receives a set of parties that won the leadership election. This set is selected by sampling a random coin for each party, weighted by their stake using the same algorithm as in Ouroboros Praos [DGKR18]. We note that while this leakage is protocol-specific, it follows a general principle of leaking the elected leaders in a protocol. Specifically, honest parties will be selected by $\mathsf{Lkg}_{\mathsf{lead}}$ with the probability of them winning a leadership election in Ouroboros Crypsinous. This probability is the same as in Ouroboros Genesis, and is the function ϕ_f of their stake, where ϕ_f is the independent aggregation function described in [DGKR18, BGK+18].

In addition to this, we note Zerocash-style protocols will allow an adaptively corrupting adversary to compute the serial number of coins *it sent* to an honest party after corrupting them. As the serial number is by necessity committing, the simulator must know when such adversarially sent coins are spent, to ensure the consistency of the simulation. For this reason, we also leak the points adversarially sent coins are spent.

Algorithm $\mathsf{Lkg}_{\mathsf{lead}}$ **for** $\mathcal{G}_{\mathrm{PL}}$

The $\mathsf{Lkg}_{\mathsf{lead}}$ algorithm maintains a record of past leaks, L_τ for each past time τ. This is to ensure the adversary is limited in accessing the leakage function for past slots.

procedure $\mathsf{Lkg}_{\mathsf{lead}}(\mathsf{state}, \mathsf{buffer}, \tau)$
  if L_τ is recorded **then return** L_τ
  Determine ep, the epoch for the time slot τ.
  Determine τ_{ep}, the time at which the stakeholder distribution for the epoch ep was frozen.
  Let $L \leftarrow \varnothing$
  for each party U_p **do**
    Determine the valid coins of U_p in $\mathsf{state}_{\tau_{ep}}$.
    Determine U_p's relative stake α_{U_p}.
    With probability of $\phi_f(\alpha_{U_p})$, add U_p to L.
  end for
  for each adversarially generated coin **c do**
    if c was spent in state or buffer **then**
      Let tx be the transaction it was spent in
      Let i be the index of the coin in the transaction
      Let $S \leftarrow S \cup \{(\mathsf{tx}, i)\}$
    end if
  end for
  Record $L_\tau \leftarrow L$, and **return** L, S

end procedure

In a preliminary step of our analysis we also utilize a leakage function leaking all information, $\mathsf{Lkg}_{\mathsf{id}}$. This is effectively the identity function, simply returning the parameters state, buffer, and τ passed to it. With this leakage the private ledger effectively becomes a standard ledger from [BMTZ17, BGK+18], with a stricter interface to the environment, as the simulator still receives all information it would with the standard non-private ledger.

7.6 The Ouroboros-Crypsinous Protocol

In this section we provide a detailed description of our protocol Ouroboros-Crypsinous as a (G)UC protocol. The protocol has a similar structure as Ouroboros-Genesis [BGK+18], but differs considerably in the leaderelection, and the processing of transactions. As already discussed, the protocol assumes access to a global random oracle and clock, and functionalities for network, encryption, and NIZK.

7.6.1 Ideal-World Transactions

Before we delve into the protocol details, we note that unlike many other ledger protocols, we assign meaning to transactions, and this meaning, while more precisely defined later on, is helpful to understand the high-level design. Specifically, we consider ideal-world transactions starting with $(\text{PUBLIC}, \text{TRANSFER})$ to be *transfer transactions*. While it may appear sufficient to have ideal-world transfers appear as something like "give 0.05 of Alice's stake to Bob", our realization of transfers using a Zerocash-like [BCG+14] design introduces some subtleties that need to be reflected in the ideal world. Specifically, we will require parties to specify *which* coins they are attempting to spend. Specifically, as in Zerocash, two coins are burned, and two coins created, in any transfer. As a special case, as our protocol has no other minting functionality, we allow a zero-value coin to be burned in place of the second coin. Formally, the transactions have the following form: $((\text{PUBLIC}, \text{TRANSFER}), (pk_r, \mathbf{c}_4), (pk_s, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3))$, where $\mathbf{c}_i$ are ID/value pairs. This can be interpreted as "transfer the coins $\mathbf{c}_1$ and $\mathbf{c}_2$ to coins $\mathbf{c}_3$ and $\mathbf{c}_4$." It is worth noting that $\mathbf{c}_3$, while being a newly created coin, is not included in the component addressed to pk_r. It should be seen as a means of returning "change" from a transaction, corresponding to its real-world usage of Bitcoin and Zerocash transactions, and should therefore also be addressed to the sending party. The validation predicate ensures the total value is preserved across the transfer, and that an ID is only spent by its generating party. IDs must originate from the ledgers GENERATE interface, otherwise they are treated as invalid.

In the real world, the design looks slightly different, following the approach of Zerocash [BCG+14]. Specifically, parties locally maintain, for each coin $\mathbf{c}$, nonces, $\rho_{\mathbf{c}}$, and commitment openings, $r_{\mathbf{c}}$, to their coins. In order to spend a coin, they reveal the deterministically derived serial number, $sn_{\mathbf{c}}$, as well as prove the existence of a valid commitment, $cm_{\mathbf{c}}$, somewhere in a Merkle tree of coin commitments. Like Zerocash, newly created coins are encrypted with the recipient party's public key, and the sending party is unable to spend them as it would require the recipient's private key to correctly generate the coin's serial number. One key difference is the design of addresses, corresponding to the Ideal-world IDs. Parties will generate a new coin public/secret key pair when given a GENERATE query, and will update their secret key after spending a coin with it.

To become a leader at a time τ, parties must prove knowledge of a path in a local Merkle tree of secret keys sk^{COIN}, labeled with τ. This path is then erased by the party, to ensure leadership proofs cannot be re-made for past slots. This Merkle tree is created during key generation, with the coin's public key being derived from the Merkle tree's root, and the time of key generation. Each leaf is a PRF of the previous leaf, to reduce storage costs. We employ standard space/time trade-offs by keeping the top of the tree stored, and recomputing parts of the bottom of the tree as needed. It is parameterized by the number of leaves R, which we leave as a system parameter, although we note it could also be defined per-user.

A users public key is derived from the root of the Merkle tree, root, and the time it was created, τ. It is eligible for leadership so long as there are still paths in the tree to prove the existence of, after which the coin must be refreshed, by spending it. We stress that this is a rare occurrence, as the assumption of honest majority relies on coins not only being held by honest parties, but also being eligible for leadership.

The protocol will take ideal transactions as an input, and construct a corresponding Zerocash-style transaction in the real world. This transaction is then broadcast as usual in a blockchain protocol. On a READ request, the irrelevant information is not returned, and only the information corresponding to the original ideal-world transaction is returned back to the requester. In addition to transfers, we note that other types of transaction are accepted in the ideal world. We note that these are not validated, however, making the real-world equivalent far simpler to construct. Specifically, we encrypt each subtransaction with the public key of the party it is addressed to. On a READ request, the ciphertexts that the requesting party can decrypt are decrypted, and all others are replaced with $\perp$.

7.6.2 Protocol overview

The protocol Ouroboros-Crypsinous assumes as hybrids a network $\mathcal{F}^{\Delta}_{\text{N-MC}}$, a non-interactive-zero-knowledge scheme $\mathcal{F}_{\text{NIZK}}$, a forward-secure encryption scheme $\mathcal{F}_{\text{FwEnc}}$, a global clock $\mathcal{G}_{\text{CLOCK}}$, a global random oracle $\mathcal{G}_{\text{RO}}$, a non-interactive equivocal commitment protocol, and a CRS used by the commitment protocol, to supply the commitment public key, $\mathcal{F}_{\text{CRS}}$.

The protocol execution proceeds in disjoint, consecutive time intervals called *slots*. As in Ouroboros Genesis, slots correspond directly to rounds given by $\mathcal{G}_{\text{CLOCK}}$. In each slot sl, the parties execute a *staking procedure* to extend the blockchain. This proceeds similarly to Ouroboros Genesis, electing *leaders* to slots, with modifications to avoid revealing more information about the leader than necessary. We note that due to network-level attacks, the adversary is able to guess with good probability *which* party is the leader. Further, due to serial numbers being revealed, and being committing, an the simulator must know when coins whose serial number the adversary could guess after corruption – specifically those sent by the adversary itself – were spent. This additional leakage can be avoided if by a paranoid party, by it immediately transferring coins to itself on receipt. Further, it is only an issue for parties which *may be corrupted*. In a hypothetical setting where the adversary could commit to not corrupting a party, this party would no longer have leakage of this kind. Similar to Ouroboros-Genesis, time is also divided into larger units, called epochs, with the distribution of stake considered for leadership purposes being frozen for each epoch.

We specify a concrete transaction system, based on Zerocash [BCG+14]. Parties hold *coins* with inherent value, and a fixed total value across the system (a restriction imposed for simplifying the analysis. Adding block rewards would be a straightforward extension). The Ouroboros Genesis leadership election is performed on a per-coin basis, with each coin competing separately. If any of a party's coins win the election, the party proceeds to generate a new block, extending their current chain. The block itself is generated as in Ouroboros-Genesis, although the validity of it is proved differently. Specifically, $\mathcal{F}_{\text{NIZK}}$ is used to produce a signature of knowledge of a coin that won the leadership election during a given slot. This proof is done in a Zerocash style, and involves renewing the coin in question. Specifically, the Zerocash serial number of the leading coin is revealed, and a new coin of the same value is minted. We also refer to this proof, together with its auxiliary information such as the spent serial number and newly created coin commitment, as a *leadership transaction*.

We note that Ouroboros-Genesis requires the stakeholder distribution to be frozen to prevent grinding attacks. In order to allow a coin to be used for leadership proofs multiple times in an epoch, we introduce a new resistance mechanism against attacks of this type: The newly generated coins in leadership transactions have their nonce deterministically derived from the nonce of the old coin. The leadership test itself utilizes only this nonce from the coin as a seed – it follows that the leadership test for the derived coin is fixed along with the randomness of the epoch.

Once a block is created, the party broadcasts the new chain, extended with this block. Further, the party broadcasts the leadership transaction separately, in order to ensure the newly created coin will eventually be valid, even if the consensus does not adopt the broadcast chain.

A chain proposed by any party might be adopted only if it satisfies the following two conditions: (1) it is valid according to a well defined validation procedure, and (2) the block corresponding to each slot has a signature of knowledge from a coin winning the corresponding slot.

To ensure the second property we need the implicit slot-leader lottery to provide its winners (slot leaders) with a certificate/proof of slot-leadership. For this reason, we implement the slot-leader election as follows: Each party U_p checks, for each of their coins $\mathbf{c}$, whether or not it is a slot leader, by locally evaluating a maliciously-unpredictable pseudo-random function, as described in Section 7.4.3, with entropy supplied by the epoch randomness η_{ep}, by being evaluated at the slot index sl and η_{ep}, seeded with the "winning coin's secret key" $\mathsf{root}_{\mathbf{c}} \parallel \rho_{\mathbf{c}}$. η_{ep} is generated similarly to Ouroboros Genesis – it is initially supplied through the CRS, then for subsequence epochs, it is sampled in a maliciously unpredictable way from "randomness contributions" ρ provided by slot leaders over the course of the previous epoch.

Specifically, we will use the MUPRF construction of Section 7.4.3, for a given group G. If the MUPRF output y is below a certain threshold $T_{\mathbf{c}}$—which depends on $\mathbf{c}$'s stake—then U_p is an eligible slot leader; furthermore, he can generate a signature of knowledge of a valid coin which satisfies these conditions. In particular, each new block broadcast by a slot leader contains a NIZK proof π, signing the rest of the block content, with the knowledge of the nonce $\rho_{\mathbf{c}}$, $sk^{\text{COIN}}_{\mathbf{c},sl}$ for the slot sl the leadership transaction is for,, proving that the nonce and secret key correspond to some unspent coin commitment $cm_{\mathbf{c}}$. The leadership transaction also *evolves* the coin that wins leadership – this is done in order to establish adaptive security, and is done by updating the coin nonce used: $\rho_{\mathbf{c}'} = \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$. A new coin, in the same value, with this updated – and, crucially, deterministic – nonce is created, and committed in the transaction. In particular, parties erase $\rho_{\mathbf{c}}$, and only maintain $\rho_{\mathbf{c}'}$ after the leadership proof is generated.

We note that, as in Ouroboros-Genesis, it is possible for multiple, or no party to be a leader of any given slot. Our protocol behaves identically to Genesis in this regard, and we utilize the same chain selection rule in our protocol.

We next turn to the formal specification of the protocol Ouroboros-Crypsinous. We note that our party management is identical to that of Ouroboros Genesis, and our protocol description follows the same modular design as Ouroboros Genesis. For brevity we will not re-state parts of the genesis protocol which remain unmodified, and we will leave precise UC specification of protocol components to Appendix 7.11.

7.6.3 Real-world Transactions

Before giving the formal specification we introduce some necessary terminology and notation. Each party U stores a local blockchain $\mathcal{C}^{U_p}_{\mathsf{loc}}$—$U_p$'s local view of the blockchain.[5] Such a local blockchain is a sequence of blocks B_i ($i > 0$) where each $B \in \mathcal{C}_{\mathsf{loc}}$ has the following format: $B = (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st})$; where $\mathsf{tx}_{\mathsf{lead}} = (\text{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$, and $\mathsf{stx}_{\mathsf{proof}} = (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi)$. Here, st is the encoded data of this block, h is the hash of the same data, sl and ep are the slot and epoch the block is for, respectively, $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk^{\text{COIN}} \parallel \tau \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}'})$ is the commitment of the newly-created coin, and $sn_{\mathbf{c}} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}^{\text{COIN}}_{sk}}(\rho_{\mathbf{c}})$ is the serial number of the coin $\mathbf{c}$, which is revealed to demonstrate the coin has not been spent. We define $\rho = \mu^{sk^{\text{COIN}}_{sl}}$, where μ is $\mathcal{G}_{\text{RO}}$ evaluated at $\text{NONCE} \parallel \eta_{ep} \parallel sl$; ρ is the randomness contribution to the next epoch's randomness, ptr is the hash of the previous block, and π is a NIZK proof of the statement LEAD (defined in Section 7.12). The component $\vec{\mathsf{stx}}_{\mathsf{ref}}$ consists of a (typically empty) vector of reference leadership transactions. These are processed *before* the leadership transaction itself is processed, and serve to allow successive leadership proofs with the same coin, even when the selected chain switches.

Ouroboros Crypsinous handles three kinds of transactions: *Leadership transactions*, such as the above $\mathsf{tx}_{\mathsf{lead}}$, *transfer transactions* $\mathsf{tx}_{\mathsf{xfer}}$, and *general-purpose transactions*. Each of these is handled separately. The transfer transactions and general-purpose transactions correspond directly to ideal-world transactions with the same behaviour. Leadership transactions by contrast exist only in the real world.

[5]For brevity, wherever clear from the context we omit the party ID from the local chain notation, i.e., write $\mathcal{C}_{\mathsf{loc}}$ instead of $\mathcal{C}^{U}_{\mathsf{loc}}$.

General-purpose transactions in the ideal world consist of a vector of subtransactions, addressed either to everyone (PUBLIC), or a specific party. The corresponding real-world transaction is a vector of the same subtransactions, which are either directly the content of the ideal world transaction, in the case of a transaction addressed to PUBLIC, or an encryption of the content using $\mathcal{F}_{\text{FwEnc}}$, to the party specified as the recipient. Upon reading the state, parties attempt to decrypt ciphertexts, and failing that, replace it with $\perp$. To disambiguate transactions, we prefix generic transactions with the label GENERIC.

The implementation of transfer transactions is more involved, as we not only want to guarantee their privacy, but also their validity. To achieve this, we replace transaction which fall into the permissible ideal-world format – which we recall, is $\mathsf{tx}^{\mathsf{ideal}}_{\mathsf{xfer}} = ((\text{PUBLIC}, \text{TRANSFER}), (pk_r, (\mathsf{id}_4, v_4)), (pk_s, (\mathsf{id}_1, v_1), (\mathsf{id}_2, v_2), (\mathsf{id}_3, v_3)))$ – with a cryptographic construction hiding the respective information. We define a real transfer transaction to be: $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}} = (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, c_r)$, where $\mathsf{stx}_{\mathsf{proof}} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root}, \pi)$, and c_r is a $\mathcal{F}_{\text{FwEnc}}$-encryption for the slot the transaction was submitted of $\mathsf{stx}_{\mathsf{rcpt}} = (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})$ to pk_r. Similar to leadership transactions, $(cm_{\mathbf{c}_3}, r_{\mathbf{c}_3}) = \mathsf{Comm}(pk^{\text{COIN}}_{pk_s} \parallel \tau \parallel v_{\mathbf{c}_3} \parallel \rho_{\mathbf{c}_3})$, and $(cm_{\mathbf{c}_4}, r_{\mathbf{c}_4}) = \mathsf{Comm}(pk^{\text{COIN}}_{pk_r} \parallel \tau \parallel v_{\mathbf{c}_4} \parallel \rho_{\mathbf{c}_4})$; $sn_{\mathbf{c}_1}$ and $sn_{\mathbf{c}_2}$ are revealed to spend the coins $\mathbf{c}_1$ and $\mathbf{c}_2$ respectively, and π proves the statement XFER (defined in Section 7.12), specifically proving the existence of $cm_{\mathbf{c}_1}$ and $cm_{\mathbf{c}_2}$, in the Merkle tree of coin commitments with the root root, as well as various consistency properties. The use of $\mathcal{F}_{\text{FwEnc}}$ implies that parties will not be able to decrypt ciphertexts addressed to them indefinitely, however they are still required to respond with the corresponding ideal-world information to READ requests. As a result, when a transfer transaction is first seen and decrypted, the corresponding ideal world transaction is locally stored. Further, parties maintain locally the information needed to spend coins they own – specifically $(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$.

7.6.4 Interacting with the Ledger

At the core of the Ouroboros Crypsinous protocol is the process that allows parties to maintain the ledger. There are three types of processes that are triggered by three different commands provided that the party is already registered to all its local and global functionalities.

- The command $(\text{SUBMIT}, \text{sid}, \mathsf{tx})$ is used for sending a new transaction to the ledger. The party maps tx to a corresponding $\mathsf{tx}^{\mathsf{real}}$, which is stored in the parties' local transaction buffer, and multicast to the network.
- The command $(\text{GENERATE}, \text{sid})$ is used for creating a new address, which can be used by other parties to transfer funds to this current party.
- The command $(\text{READ}, \text{sid})$ is used for the environment to ask for a read of the current ledger state. On receipt, the party maps each transaction $\vec{\mathsf{st}}^{\lceil k}$ to its ideal-world equivalent, and returns this ideal-world chain.
- The command $(\text{MAINTAIN-LEDGER}, \text{sid})$ triggers the main ledger update. A party receiving this command first fetches from its network all information relevant for the current round, then it uses the received information to update its local info—i.e., asks the clock for the current time τ, updates its epoch counter ep, its slot counter sl, and its (local view of) stake distribution parameters, accordingly; finally it invokes the staking procedure unless it has already done so in the current round. If this is the first time that the party processes a $(\text{MAINTAIN-LEDGER}, \text{sid})$ message then before doing anything else, the party invokes an initialization protocol to receive the initial information it needs to start executing the protocol—in particular the genesis block.

The relevant sub-processes involved in handling these queries are detailed in the following sections. After introducing each of these basic ingredients, we conclude with a technical overview of the main ledger maintenance protocol LedgerMaintenance, a detailed specification of the protocol ReadState for answering requests to read the ledger's state, and a detailed specification of the protocols SubmitXfer and SubmitGeneric.

Party Initialization A party that has been registered with all its resources and setups becomes operational by invoking the initialization protocol Initialization-Crypsinous upon processing its first command. As a first step the party receives its encryption key from $\mathcal{F}_{\text{FwEnc}}$. It receives any initial stake it may have as a single coin from $\mathcal{F}_{\text{INIT}}$. Subsequently, protocol Initialization-Crypsinous proceeds as in Ouroboros-Genesis, although it does

not register any keys. This is managed instead by the ledgers GENERATE interface. The precise description of the initialization can be found in Section 7.11.

The Staking Procedure The next part of the ledger-maintenance protocol is the staking procedure which is used for the slot leader to compute and send the next block. A party U_p is an eligible slot leader for a particular slot sl in an epoch ep if, one of U_p's coins, $\mathbf{c}$, is both eligible for leadership in ep, and a PRF-value depending on sl and the coin nonce $\rho_{\mathbf{c}}$ and secret key sk_{τ}^{COIN}, is smaller than a threshold value $T_{\mathbf{c}}$. We discuss when a coin is considered eligible for leadership, and how its threshold is determined. A coin is eligible for leadership depending on when, and how, its corresponding commitment entered the chain. Specifically, if its corresponding commitment was created in a transfer transaction, it is valid in a similar way as transactions are considered for leadership in an epoch: If it is sufficiently old by the time the epoch starts, it is taken as part of the snapshot fixing the stake distribution for ep. Commitments originating from leadership transactions are always immediately eligible for leadership, as their nonce and secret key are deterministically derived. It is possible, although unusual, for the leadership transaction a coin originates from to not be present in the chain the party is currently attempting to extend. In this case, the coin is *still eligible*, as the originating leadership transaction will be added to $\vec{\mathsf{stx}}_{\mathsf{ref}}$.

Each coin $\mathbf{c}$'s value $v_{\mathbf{c}}$ induces a relative stake for the coin, $\alpha_{\mathbf{c}}$. We use the same function $\phi_f(\alpha_{\mathbf{c}})$ to determine the probability of a coin winning the leadership election, with the corresponding threshold, $T_{\mathbf{c}} = \mathsf{ord}(G)\phi_f(\alpha_{\mathbf{c}})$. Due to the independent aggregation property of ϕ_f, the probability of a party winning the leadership election in Crypsinous and in Genesis is initially the same, regardless of how it is split between coins. One key difference, however, is that when a coin is *transferred* in Crypsinous, it is *no longer eligible for leadership*. As a direct consequence, any stake transferred during an epoch must be considered adversarial for the given epoch.

The technical description of the staking procedure can be found in Section 7.11.2. It evaluates two district MUPRFs for each eligible coin. If the output of one of these is under the target for some coin, the party is a slot leader, and continues to create a new block B from their current transaction buffer. Aside of the main contents, the party assembles a leadership transaction and assigns it to the block. This transaction includes a NIZK proof of leadership – specifically of the statement LEAD – and acts as a signature of knowledge over the block content, as well as the pointer to the previous block. An updated blockchain $\mathcal{C}_{\mathsf{loc}}$ containing the new block B is finally multicast over the network.

From the staking procedure we construct the ledger maintenance protocol, which in addition to attempting to stake on each block, monitors incoming transactions and chains, decrypts ciphertexts where possible, updates the parties local state by adding received coins, records received messages, and performs the chain selection of [BGK+18]. The full description can be found in Section 7.11.3

Submitting Transactions Transactions submitted to the Ouroboros Crypsinous protocols are, as previously discussed, first mapped to corresponding real-world transactions, which then get handled as standard ledger transactions by being broadcast over a multicast network, and assembled into blocks. Specifically, transfer transactions are mapped to Zerocash-like transactions, where only the first coin received to a given address it spent, and other transactions are mapped into encrypted components. The submitting procedure for transfer transactions is described in Section 7.11.4, and that for generic transaction in Section 7.11.5.

Reading the State The last command related to the interaction with the ledger is the read command $(\text{READ}, \text{sid})$ that is used to read the current contents of the state. Note that in the ideal world, the result of issuing such a command is for the ledger to output a (long enough prefix) of the ideal-world state of the ledger, with parts the party does not have access to being hidden. As the format of real-world transactions differs, we need to invert the map from real transactions to the corresponding ideal transactions. For generic transactions, this is a little tricky, as the use of forward-secure encryption implies that the information associated with the transaction in the ideal world is erased in the real world. To circumvent this, parties maintain a log, recording information necessary to reconstruct the ideal-world representation of the transaction. The full description of this reconstruction can be found in Section 7.11.6.

7.6.5 Transaction Validity

Transaction validity again differs in the real and ideal world, as the transactions themselves differ.

Ideal World Validation The ideal world validation predicate validates only transfer transactions. It is parameterized by the initial distribution of coins $\mathfrak{C}_1$. It maintains, for each ID, an ordered sequence of received values, the ID's owner, and a flag marking whether the ID has already been used for spending. For each transfer transaction validated, first its format is enforced. Next, it asserts that $v_1 + v_2 = v_3 + v_4$. It checks that the IDs of $\mathbf{c}_1$ and $\mathbf{c}_2$ have indeed received transfer of value v_1 and v_2 respectively (and, if the IDs and value are equal, have received at least *two* transfers of that value). If there is ambiguity as to which coins to spend, those received first are spent. As a special case, if the ID of $\mathbf{c}_2$ is $\perp$, and $v_2 = 0$, it is always valid.[6] It is further checked that the coins the party is trying to spend are "old enough", specifically, they must be in the parties local view of the ledger state. (The validation predicate has access to the parties state pointer). If the sending party is honest, we further restrict it to only spending coins to which it owns the ID. Further, honest parties must address $\mathsf{stx}_{\mathsf{chng}}$ to their own public key – i.e. the first value generated by $(\text{GENERATE}, \text{sid}, \text{ID})$ by the party. If the sending party is corrupted, it may spend the coins of other corrupted parties, as well as arbitrary received values. If other transactions in the *buffer* attempt to spend the same coins, and the transaction is honest, it is also rejected – as in this case the party is attempting to double spend and de-anonymize themselves.

Finally, if the transaction is valid, a new receipt of a value of v_3 is recorded for $\mathbf{c}_3$, and respectively with v_4, and $\mathbf{c}_4$. The values spent are erased from the values lists of $\mathbf{c}_1$ and $\mathbf{c}_2$'s IDs, and their "spent" flags are set (with the exception of the id $\perp$).

Real-world Validation The real-world validation predicate maintains three sets, the sets of coin commitments $\mathbb{C}^{\mathsf{spend}}$, $\mathbb{C}^{\mathsf{lead}}$ for spending and leadership respectively, initialized to the initial set of coin commitments $\mathbb{C}_1$, and the set of spent serial numbers $\mathbb{S}$, initialized to $\varnothing$. A chain is validated transaction by transaction. Leadership transactions and transfer transactions are both validated, other transactions are ignored. A leadership transaction is valid iff all leadership transactions in $\vec{\mathsf{stx}}_{\mathsf{ref}}$ are valid adopted leadership transactions, and the NIZK proof is valid with respect to the Merkle root of the current tree, with these adopted transactions inserted, as well as η_{ep}, and it has a greater slot number than the previous slot. Further, the serial number sn revealed in it must not be in the current $\mathbb{S}$. The root used must either be the root of the predecessor block, or the root of a past leadership transaction's Merkle tree, with only this transactions commitment added to the tree. Finally, ptr must be the hash of the previous block, and h must be the hash of the remaining transactions. After it is successfully validated, $\mathbb{S} \leftarrow \mathbb{S} \cup \{sn\}$, $\mathbb{C}^{\mathsf{lead}} \leftarrow \mathbb{C}^{\mathsf{lead}} \cup \{cm\}$, $\mathbb{C}^{\mathsf{spend}} \leftarrow \mathbb{C}^{\mathsf{spend}} \cup \{cm\}$.

Transfer transactions are likewise validated by checking the NIZK proof with respect to the public transaction component. Further, it is checked that root was at some point the root of $\mathbb{C}^{\mathsf{spend}}$, and that $\{sn_1, sn_2\} \cap \mathbb{S} = \varnothing$. If so, the effect is updating $\mathbb{S} \leftarrow \mathbb{S} \cup \{sn_1, sn_2\}$, and $\mathbb{C}^{\mathsf{spend}} \leftarrow \mathbb{C}^{\mathsf{spend}} \cup \{cm, cm_3\}$. Finally, at the start of an epoch, old enough spending coins are allowed for leadership proofs: $\mathbb{C}^{\mathsf{lead}} \leftarrow \mathbb{C}^{\mathsf{lead}} \cup \mathbb{C}^{\mathsf{spend}}_{t-k}$, where $\mathbb{C}^{\mathsf{spend}}_{t-k}$ is the set of spending coin commitments k slots before the start of the epoch.

If a leadership transaction is included normally in a block, or included in $\vec{\mathsf{stx}}_{\mathsf{ref}}$ (i.e. it is not *this block's* leadership transaction), it is considered an *adopted leadership transaction*. The validity criteria for these are different, requiring only that the proof is valid, the serial numbers are unspent, and the Merkle root was a valid root for $\mathbb{C}^{\mathsf{lead}}$ at some point. The effects of the transaction remain the same, although is is no longer the leader of a block. A block's transactions are validated *prior* to the leadership transaction, as this may depend on adopted leadership transactions. The Merkle tree root of $\mathbb{C}^{\mathsf{lead}}$ of any adopted leadership transactions chain's is saved and preserved. These are valid for other leadership transactions in the same epoch. Specifically, they are also valid for the leadership transaction of the block it is contained in.

Generic transactions are valid if and only if they do not start with the symbol $(\text{PUBLIC}, \text{TRANSFER})$.

[6]This permits parties with only one coin to spend it.

7.7 Security Analysis

We split our security analysis of Ouroboros-Crypsinous into two parts: In a first, warm-up part, we show that Ouroboros-Crypsinous realizes a "non-private" version of $\mathcal{G}_{\text{PL}}$ – specifically, we show that it realizes $\mathcal{G}_{\text{PL}}$ with Lkg set to the identity function $\mathsf{Lkg}_{\mathsf{id}}$; i.e. the ledger leaks its entire content to the simulator, described in detail in Section 7.10. We argue that the simulator $\mathcal{S}_1$ can simulate any real-world attacks on Ouroboros-Crypsinous against a non-private $\mathcal{G}_{\text{PL}}$. This first part already proves that our protocol satisfies all the properties of the public ledger, including chain quality, common prefix, and chain growth. In a second part, we argue that in addition to the above, it also satisfied privacy. This is done by instantiating Lkg to $\mathsf{Lkg}_{\mathsf{lead}}$, in which only the leaders of a given slot are leaked. For this case we provide a simulator $\mathcal{S}_2$ who is able, with access only to this restricted leakage to simulate the outputs of $\mathcal{S}_1$. generate a view which is indistinguishable from $\mathcal{S}_1$.

Theorem 17. Ouroboros-Crypsinous*, in the* $(\mathcal{W}_{\text{OC}}^{PoS}(\mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{LEAD}}}, \mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{XFER}}}, \mathcal{F}_{\text{FwEnc}}, \mathcal{F}_{\text{N-MC}}^{\Delta}), \mathcal{G}_{\text{RO}}, \mathcal{G}_{\text{CLOCK}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\text{PL}}$ *with* $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$*, under the DDH assumption.*[7]

Proof (sketch). The backbone of the proof of Theorem 17 is similar to the security proof of Ouroboros Genesis [BGK+18] with some surgical modifications; in particular, in Step 1 we argue that the usage of NIZKs, nonces, and key-private forward-secure encryption, can replace the usage of forward secure signatures, and in Step 2 we argue that the usage of NIZKs and MUPRFs can replace the usage of VRFs in Genesis. In a nutshell, this allows us to argue in Step 3 argue that leadership transactions in Crypsinous can be used to replace leadership proofs in Genesis. This allows us to leverage the security analysis from Ouroboros Genesis [BGK+18] in Step 4 for proving that Crypsinous implements, at the very least, a non-private version of the ledger.

Transactions submitted to Crypsinous are pre-processed, before being handled as a Genesis transaction would be, and on reading from the ledger, this pre-processing is partially inverted. This inversion being only partial is what will later be used to establish the privacy properties of Crypsinous. In Step 5, we establish that this pre- and post-processing has the same effect as blinding a transaction in the ideal world, and that the validation predicate of Ouroboros-Crypsinous – which is run only against pre-processed transactions – is equivalent to its ideal-world counterpart. Finally, in Step 6, we argue that combined, these properties demonstrate realisation of $\mathcal{G}_{\text{PL}}$ with $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$.

Step 1. The security properties guaranteed by $\mathcal{F}_{\text{KES}}$, and used in [BGK+18], are those of forward-secure unforgeability, correctness, and authenticity. A proof of LEAD gives the former two properties, and a notion of authenticity that is different to $\mathcal{F}_{\text{KES}}$, but sufficient for how it is used in [BGK+18]. Non-malleable NIZKs, such as the ones used in our construction, can be interpreted as "signing" their public inputs with the knowledge of a witness [GM17]. In particular, if the witness itself contains a secret key known only to one party, a NIZK over such a witness effectively acts as a signature. In Ouroboros Crypsinous, the usage of sk^{COIN} in the witness for leadership proof effectively acts as a signature over the rest of the block, providing unforgeability, and correctness guarantees. Further, as the statement LEAD has the same conditions as a leadership proof in [BGK+18], the desired authenticity property is also satisfied. This is not sufficient to emulate $\mathcal{F}_{\text{KES}}$, however using $sk_s^{\text{COIN}}l$ and $\rho_{\mathbf{c}}$ in the witness rectifies this. As honest parties update both $sk_s^{\text{COIN}}l$ and $\rho_{\mathbf{c}}$ after the proof, and $sk_s^{\text{COIN}}l$ and $\rho_{\mathbf{c}}$ are necessary to generate a new proof for the same slot, the adversary will be unable to create leadership proof for past slots. While this is effective only so long as $sk_s^{\text{COIN}}l$ and $\rho_{\mathbf{c}}$ cannot be retrieved from elsewhere. $sk_s^{\text{COIN}}l$ is generated locally by an honest party, is never communicated by it (except to $\mathcal{F}_{\text{NIZK}}$, which guarantees its secrecy), and is erased by the honest party in the same slot.

Step 2. The property of VRF provability is directly captured by the correctness of NIZKs, and that of uniqueness is directly captured by non-malleability. Pseudorandomness is directly supplied by the security under malicious key generation of MUPRFs. Two VRF calls are embedded in the NIZK; the VRF used to generate the randomness

[7] We will be working under this assumption throughout the rest of the security analysis, and will typically leave it implicit. We will also be assuming the binding (under discrete log, which is implied by DDH), and hiding of our commitments, and the pseudo-randomness of our PRFs implicitly.

contribution ρ, and the VRF used to check the target. While in Ouroboros Crypsinous the latter is not publicly revealed, it is still present, and is verified by a verification of the NIZK. The NIZK is not as flexible as the VRF, in that it cannot be used to generate arbitrary VRF proofs at any time, however this is simply as the verification is stricter. The NIZK inputs in Ouroboros Crypsinous depend on the coin secret key, while in Ouroboros Genesis, they depend on the *party's* secret key. As Ouroboros Genesis anticipates parties acting as multiple parties in the protocol, we can simply consider each Crypsinous coin as one Genesis party.

Step 3. A leadership transaction in Ouroboros-Crypsinous can be made only if a coin passes the same threshold check as in Ouroboros-Genesis. Due to the independent aggregation property of the threshold function, the probability of this happening for a party holding a specific value of (honest) stake is equal in Crypsinous and Genesis. Furthermore, the NIZK ensures the impossibility of creating a leadership transaction *without* winning this election in Crypsinous, while the VRF validation, and block validity check enforce the same property in Genesis. The mechanism of "adopted" leadership transaction ensures this property is preserved, even by a party selecting a new local chain.

Due to the equivalent output distribution of VRFs and PRFs in Genesis and Crypsinous respectively, the randomness contribution ρ is also equivalent.

Step 4. Given we can replace leadership proofs with leadership transactions in the $\mathcal{G}_{\text{LEDGER}}$ proof of [BGK+18], the rest of the proof can be carried out the same for Ouroboros-Crypsinous. This establishes that, Ouroboros-Crypsinous effectively runs an internal ledger. While the transactions posted to this ledger are not directly those posted to Ouroboros-Crypsinous itself, we will establish their relationship, and that this corresponds directly to the difference between the public and private ledger.

Step 5. Submitted transactions are pre-processed before being sent to the network, and transactions from the network are post-processed on a READ request in Ouroboros-Crypsinous. For brevity, we will refer to the former mapping as f, and the latter as $f_{U_p}^{-1}$. We define *consistency* of this mapping to be two the following two properties things: First, a validation predicate – specifically instantiated to that of Ouroboros-Crypsinous – over the mapped transactions must exists that holds if and only if the ideal-world validation predicate over the original transactions holds. Second, $f_{U_p}^{-1} \circ f = \mathsf{BlindTx}(\{U_p\})$ – i.e. READ requests return the same as $f_{U_p}^{-1}$ of the READ in the mapped ledger. Specifically, as the real-world validation predicate already operates on the mapped transactions, this predicate should behave the same as the ideal-world predicate over the original transactions.

For generic transactions, this is straightforward: subtransactions addressed to PUBLIC are preserved, and not affected by the mapping. Subtransactions addressed to a party U_p are encrypted with pk_p^{ENC} in the real world, and each party attempts to decrypt them on the inverse mapping. Specifically, subtransactions addressed to any other party will fail to decrypt, and be replaced with $\perp$, while subtransactions which are correctly encrypted, will be replaced with (pk_p, M), where M is the originally encrypted plaintext. This matches the behaviour of BlindTx exactly. Finally, the validation predicate is always true for generic transactions in the ideal world, and is only false for generic transactions that start with TRANSFER in the real world – which have no ideal world equivalent, and should cannot be created by honest parties.

Transfer transactions This leaves us with the consistency of mappings for transfer and leadership transactions. In addition to being standard transactions, transfer transactions induce a stakeholder distribution. They are intrinsically linked with leadership transactions in the real world, so we will consider these as well. The ledgers, both real and ideal, can be read as a sequence of transfer-, and in the real world leadership- transactions. We will prove by induction that validity is equivalent in the real and ideal world, as well as that the inverse mapping of the real-world transaction is the ideal transaction. First, we note the induction hypothesis: For every vector of transfer and (in the real world) leadership transactions in the real and ideal worlds, two sets of valid coins are induced: a) The set of valid ideal-world coins, where each coin has a party, ID (which the simulator sets to be the coin public key $pk_{\mathbf{c}}^{\text{COIN}}$), and value, and b) The set of valid real-world coins, which have the same attributes, as well as an associated coin secret key sk^{COIN}, a nonce $\rho_{\mathbf{c}}$, and a commitment randomness $r_{\mathbf{c}}$. The induction hypothesis is that these sets are equivalent, i.e. the ideal set is equal to the real set without the secret key, nonce

and randomness, and that in the vector of transactions, the same transfer transactions were considered valid in both worlds.

As a base case, this is guaranteed by $\mathcal{F}_{\text{INIT}}$, which creates the same distribution of coins in the real world as was given in the ideal world, selecting random $\rho_{\mathbf{c}}$ and $r_{\mathbf{c}}$ values. In the induction step, we increase the real-world transaction vector by one transaction. There are four cases, depending on whether the transaction is honest or adversarial, and whether it is a transfer, or leadership transaction. We will consider the honest cases first.

Honest leadership In the case of an honest leadership transaction, the transaction is valid in the real world, as honest parties would not post an invalid transaction. It spends a coin, and recreates a coin of the same value. This is reflected by updating the set of real-world coins by replacing $\rho_{\mathbf{c}}$ and, $r_{\mathbf{c}}$ with new values $\rho_{\mathbf{c}'}$, and $r_{\mathbf{c}'}$. Trivially, this maintains the induction hypothesis.

Honest transfers In the case of an honest transfer transaction, the ideal world transaction is valid iff the two spent coins were the first coins received at an ID owned by the sending party, the transaction is zero-sum, and the address of the "change" coin is also owned by the same party. If these conditions do *not* hold, the honest party would ignore the request in the real world. If they do, the honest party is, by induction hypothesis, guaranteed to know the corresponding sk_{τ}^{COIN}, $\rho_{\mathbf{c}}$ and $r_{\mathbf{c}}$-values of the coins that are spent, so it is able to generate a valid transaction and NIZK proof. Afterwards, in the real and ideal world, the coin is removed from the set of valid coins, and the newly created coins are not yet added, but will be added once the transaction has been confirmed. We conclude the induction hypothesis is maintained.

Adversarial transactions To consider adversarial transactions, the simulator does not immediately add them to the buffer. Instead, the simulator locally stores them, and waits until the adversary has them sufficiently deep in the chain that they must be added to the ideal world state. At this stage, the simulator adds them to the ideal-world buffer, and immediately promotes them to the state. This allows the simulator to manage conflicting adversarial transactions, as it simply waits for the adversary itself to resolve the conflict. In particular, transactions attempting to spend the same coin, in either a leadership or transfer transaction, will be conflicting, as they would reveal the same serial number. Once and adversarial leadership transaction is confirmed in the same way, the adversary will control the same updated coins as in the honest case, and will be unable to use the old coins again, as the validation predicate will detect and block the reuse of the coins serial number.

Adversarial transfers As the simulator waits until it enters the state, we need only consider sufficiently deep, valid transactions in the real world, and ensure the simulator can create a corresponding ideal world transaction. The real-world transaction will need to spend two valid coins, which can originate only from corrupted parties. It creates two new coins, addressed to any party, or potentially no party at all, of the same value. This directly corresponds to a legal adversarial transaction in the ideal world, and by induction hypothesis, all coins spent will be unused. The adversary cannot spend honest coins, as it does not know their secret key, with which to create a NIZK proof, cannot spend coins multiple times, as this would invalidly reveal the same serial number twice. Finally, it cannot spend non-existent coins, as it could not provide a Merkle path witness.

Equivalence We conclude that real and ideal transactions induce the same set of valid coins, and are valid in the same cases. The simulator delaying adversarial transactions in the ideal world is not visible to the environment in any way, as the buffer is only seen by the simulator itself, and the validation predicate (which does not care about the order of adversarial transaction until they enter the state). The set of coins induces a stakeholder distribution, as required by the proof of [BGK$^+$18].

Finally, the inverse mapping of parties views correspond to their ideal-world views. Specifically, if the party sees *anything* in the ideal world, it is the recipient of a coin, in which case it it need only be able to supply $pk_{\mathbf{c}}^{\text{COIN}}$ and $v_{\mathbf{c}}$ in the ideal world – provided the coin has not since been spent. If the transaction was honest, the party will have seen them on decrypting its ciphertext and – iff the coin has not been spent – can be found recorded in log. If the transaction is dishonest, either the ciphertext still correctly encrypts the coin, or, if it does not, the ideal transaction would not have been addressed to the honest party, but to the adversary instead. We conclude that honest parties response to READ requests in the real and ideal worlds match.

Step 6. The private ledger differs primarily from the standard ledger in that it a) applies Blind to the output of READ requests, b) leaks less information to the adversary, and c) provides a mechanism for unique ID generation (which are used internally). Difference a) follows directly from the consistency demonstrated in Step 5. Further, we are considering an overly permissive leakage predicate, $\mathsf{Lkg}_{\mathsf{id}}$, which provides the adversary with the same information it would receive from the standard ledger satisfying b). Finally, Ouroboros Crypsinous allows ID generation, which are generated as either PRF outputs of a PRF seeded with a random, secret value, which will lead to unique IDs for honest parties with overwhelming probability, $\mathcal{F}_{\text{FwEnc}}$ public keys, which are guaranteed uniqueness, or randomly samples values from $\{0,1\}^\kappa$, which have a negligible probability of collision. We conclude that $\mathsf{Ouroboros\text{-}Crypsinous}$ realizes $\mathcal{G}_{\text{PL}}$ with $\mathcal{S}_1$, under the leakage predicate $\mathsf{Lkg}_{\mathsf{id}}$. □

Theorem 18. $\mathsf{Ouroboros\text{-}Crypsinous}$*, in the* $(\mathcal{W}_{\text{OC}}^{PoS}(\mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{LEAD}}}, \mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{XFER}}}, \mathcal{F}_{\text{FwEnc}}, \mathcal{F}_{\text{N-MC}}^{\Delta}), \mathcal{G}_{\text{RO}}, \mathcal{G}_{\text{CLOCK}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\text{PL}}$ *with* $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{lead}}$ *under the DDH assumption.*

Proof (sketch). The leakage $\mathsf{Lkg}_{\mathsf{lead}}$ leaks only the leader of any given slot. We utilize a modified version of $\mathcal{S}_1$, which differs only in that it creates simulated transaction instead of real transactions, and reconstructs a corrupted party's state when required. The modified simulator, $\mathcal{S}_2$ is described in detail in Section 7.10.2. In Step 1, we argue that the simulated transactions are indistinguishable from real transactions, and in Step 2, we argue that the reconstructed party state is indistinguishable from a real party's state. Finally, in Step 3, we argue that the simulator $\mathcal{S}_2$ is indistinguishable from $\mathcal{S}_1$, although requiring less leakage from the private ledger functionality. As a result, the same security argument as for $\mathcal{S}_1$ holds with respect to $\mathcal{G}_{\text{PL}}$ with restricted leakage.

Step 1. There are three primitives that are simulated in simulated transactions: Commitments, NIZKs, and $\mathcal{F}_{\text{FwEnc}}$ encryptions. Due to the simulation security of NIZKs, and the equivocality of the commitments, we know they are indistinguishable from real NIZKs and commitments respectively. For $\mathcal{F}_{\text{FwEnc}}$, the simulator hands the adversary the same information about the plaintext (namely, the length) as the functionality itself, leaving the adversary with no information to distinguish. As transactions consist of these primitives, and the simulator accurately knows the format and originating party of a transaction, it can create a perfect simulated equivalent of the transaction, and broadcast it on behalf of the same party.

Step 2. While the first simulator was effectively running the protocol for real parties, making corruption trivial, $\mathcal{S}_2$ must reconstruct the parties local state in a way the adversary cannot distinguish from a real execution. Parties maintain four important state variables: the local chain, $\mathcal{C}_{\mathsf{loc}}$, the local buffer buffer, the set of coins $\mathfrak{C}$ (as well as $\mathfrak{C}_{\mathsf{free}}$, and $\mathfrak{C}_{\mathsf{cnd}}$), and the log of transfer interactions, and ciphertext to plaintext mappings, log. Maintaining $\mathcal{C}_{\mathsf{loc}}$, and buffer is straightforward, as the network interactions directly dictate their content, and the network is not anonymous. This leaves as the only major issues the reconstruction of $\mathfrak{C}$, $\mathfrak{C}_{\mathsf{free}}$, $\mathfrak{C}_{\mathsf{cnd}}$, and log. When a real-world party's corruption is requested, the simulator corrupts the corresponding ideal-world party. This allows the simulator to extract when the party received, transfers in the ideal world, all of which are guaranteed to be unspent, as well as the plaintexts corresponding to the ciphertext of subtransactions addressed to the party. At these points, a transfer, or generic transaction will have also been made in the real world. This transaction is either a real transaction, in which case the simulator can extract its content from its simulated $\mathcal{F}_{\text{FwEnc}}$. The corrupted party can only be the recipient U_r of such transactions (as this is the only party which may read it). There is one commitment in the transaction, that is created for a new coin of this party, and one encrypted $\mathcal{F}_{\text{FwEnc}}$ message that encrypts the corresponding secret values used to control it. The simulator randomly samples $\rho_{\mathbf{c}} \xleftarrow{\$} \{0,1\}^{\ell}_{\mathsf{PRF}}$, and retrieves $pk_{\mathbf{c}}^{\text{COIN}}$, $v_{\mathbf{c}}$ from the corresponding ideal-world transaction. As the ideal-world transaction is valid, we know $pk_{\mathbf{c}}^{\text{COIN}}$ must be a valid ID for the corrupted party, in which case the simulator provided it, and knows the corresponding secret key $sk_{\mathbf{c}}^{\text{COIN}}$. It then opens the commitment $cm_{\mathbf{c}}$ to $pk_{\mathbf{c}}^{\text{COIN}} \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}}$, with the opening randomness $r_{\mathbf{c}}$. This allows the simulator to populate $\mathfrak{C}$, $\mathfrak{C}_{\mathsf{free}}$, and $\mathfrak{C}_{\mathsf{cnd}}$ with coins generated by transfer transactions, depending on their stage of confirmation. We further note that the $\mathcal{F}_{\text{FwEnc}}$ ciphertext can now be opened to the appropriate encryption if necessary. Finally log is populated, by recording the corresponding log action for each of these transactions.

This almost completes the simulator, with the exception of how to handle coins that were used in leadership proofs. Recall that the simulator is aware of which slots the newly-corrupted party was a leader. It is not, however,

aware of which coin won in these slots. For each leadership proof of the corrupted party, the simulator computes the probability of each of the party's coins being the winning coin in the given slot, and samples from this distribution a single coin $\mathbf{c}$. It then ensures this coin is appropriately updated – computing $sk_{\mathbf{c}'}^{\text{COIN}} = \mathsf{PRF}^{\mathsf{evl}}_{sk_{\mathbf{c}}^{\text{COIN}}}(1)$, and $\rho_{\mathbf{c}'} = \mathsf{PRF}^{\mathsf{evl}}_{sk_{\mathbf{c}}^{\text{COIN}}}(\rho_{\mathbf{c}})$, opening $cm_{\mathbf{c}'}$, the commitment in the corresponding real-world leadership proof to $pk_{\mathbf{c}'}^{\text{COIN}} \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}'}$, with the resulting randomness being $r_{\mathbf{c}}$. This is added to $\mathfrak{C}$, with the preimage being removed. As the adversary cannot find the preimage of $sk_{\mathbf{c}'}^{\text{COIN}}$, or $\rho_{\mathbf{c}'}$, the adversary cannot perform consistency checks involving the previous coin, such as checking serial numbers match what they should.

As the state of the party handed to the simulator is correct, and any sampled value in it are either purely random, or originates from the equivocal commitment scheme, the adversary cannot distinguish the corrupted parties state from the real parties state.

Step 3. We conclude from Theorem 17, and our observations in Steps 1 and 2, combined with the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ differ only in simulating transactions and corruption, that Theorem 18 holds. □

7.8 Performance Estimation

Coin transfers are modeled after Zerocash's [BCG+14] pour transactions. This enables us to reuse much of the existing implementation work invested on optimizing the performance critical SNARK operations by the Zcash project, cf. [HBHW18].

Like Zerocash, our transfer transactions pour two old coins into two new coins. In contrast, a leadership transaction only updates a single coin. The additional costs incurred are two evaluations of a PRF to compute ρ_{c_2} and $sk_{\mathbf{c}_2}^{\text{COIN}}$ for updating the coin in a deterministic manner, two evaluations of MUPRF, and one range-proof to determine the winners of the leadership election lottery. We approximate ϕ_f using a linear function as in Bitcoin. The PRF is implemented using a SHA256 compression function. The MUPRF requires variable base group exponentiations. As we require equivocal commitments, we replace the SHA256 coin commitments of Zerocash that require 83,712 constraints with the Pedersen commitments of Sapling [HBHW18] which require only approximately 2,542 constraints. Purely for performance reasons, we also replace the original SHA-256 Merkle tree of Zerocash with the Pedersen hash-based tree used in Sapling.

In total, see Table 7.2, the multiplication count of a leadership SNARK relation is less than a transfer relation by about 42K constraints. Furthermore, the number of constraints used by our transfer relations is within a small margin of those used in an equivalent Sapling transfer relation. While have not focused on optimizing this process as Sapling has, by parallelizing the NIZK proofs, we emphasize that even unoptimized, Ouroboros Crypsinous would have a proving time only around double that of Sapling.

Primitive	Approx. constraints
SHA256	27,904
Exponentiation (variable base)	3,252 ([HBHW18], page 128)
Hidden range proof	256
Pedersen commitment	1,006 + 2.666 per bit[8]

Table 7.1: Number of multiplicative constraints in SNARK relations

We note in passing that the forward-secure encryption scheme is only needed for transfers and does not affect the SNARK relations we need to prove which is dominating performance. Likewise, the usage of a simulation secure NIZK will increase proving time, and proof lengths. Nevertheless, in both cases, the performance penalty

[8] https://github.com/zcash/zcash/issues/2634

Constraint count	$\mathcal{L}_{\text{XFER}}$	$\mathcal{L}_{\text{LEAD}}$
Check $pk_{\mathbf{c}_i}^{\text{COIN}}$	$2 \times 27,904$	$27,904$
Check $\rho_{\mathbf{c}_2}$, $sk_{\mathbf{c}_2}^{\text{COIN}}$		$2 \times 27,904$
Path for $cm_{\mathbf{c}_i}$	$2 \times 43,808$	$43,808$
(1 layer of 32)	$(1,369)$	$(1,369)$
Path for $\mathsf{root}_{sk_{\mathbf{c}_i}^{\text{COIN}}}$		$34,225$
(1 layer of 24)		$(1,369)$
(leaf preimage)		$(1,369)$
Check $sn_{\mathbf{c}_i}$	$2 \times 27,904$	$27,904$
Check $cm_{\mathbf{c}_i}$	$4 \times 2,542$	$2 \times 2,542$
Check $v_1 + v_2 = v_3 + v_4$	1	
Ensure that $v_1 + v_2 < 2^{64}$	65	
Check y, ρ		$2 \times 3,252$
Check (approx.) $y < \mathsf{ord}(G)\phi_f(v)$		256
Total	209,466	201,493

Table 7.2: Number of constraints per SNARK statement

is not intrinsic to the POS setting and it would equally affect a POW-based protocol like Zerocash if one wanted to make it simulation-secure in the adaptive corruption setting.

A second performance concern may be the cost of maintaining and updating Merkle trees of secret keys. There is a trade-off here – larger trees are more effort to maintain and use, while smaller ones may have all their paths depleted and hence require a refresh in the sense of moving the funds to a new coin. For a reasonable value of $R = 2^{24}$, this is of little practical concern. Public keys are valid for 2^{24} slots – approximately five years – and employing standard space/time trade-offs, key updates take under 10,000 hashes, with less than 500kB storage requirement. The most expensive part of the process, key generation, still takes less than a minute on a modern CPU.

7.9 Hybrid World Functionalities

Functionality $\mathcal{F}_{\text{INIT}}$

The functionality $\mathcal{F}_{\text{INIT}}$ is parameterized by the number of initial stakeholders n and their respective stakes $s_1, \ldots, s_n$. $\mathcal{F}_{\text{INIT}}$ interacts with stakeholders $U_1, \ldots, U_n$ as follows:

- In the first round, upon a request from some stakeholder U_i of the form $(\mathsf{claim}, \mathsf{sid}, U_i)$, then $\mathcal{F}_{\text{INIT}}$ samples sk^{COIN} as Ouroboros-Crypsinous does on GENERATE requests, $\rho_{\mathbf{c}_i}$ randomly, computes $pk^{\text{COIN}} \leftarrow \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}^{\text{COIN}}_{sk}}(0)$, and commits $(cm_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \mathsf{Comm}(pk^{\text{COIN}} \parallel s_i \parallel \rho_{\mathbf{c}_i})$, and returns the tuple $(pk^{\text{COIN}}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, s_i)$, along with sk^{COIN}. One all parties have registered, it samples and stores a random value $\eta_1 \xleftarrow{\$} \{0,1\}^{\lambda}$. It then constructs a genesis block $(\mathbb{C}_1, \eta_1)$, where $\mathbb{C}_1 = \{cm_{\mathbf{c}_1}, \ldots, cm_{\mathbf{c}_n}\}$.
- If this is not the first round then do the following:
  - If any of the n initial stakeholders has not send a request of the above form, i.e., a $(\mathsf{keys}, \mathsf{sid}, U_i, pk_i^{\text{ENC}})$-message, to $\mathcal{F}_{\text{INIT}}$ in the genesis round then $\mathcal{F}_{\text{INIT}}$ outputs an error and halts.
  - Otherwise, if the currently received input is a request of the form $(\mathsf{genblock_req}, sid, U_i)$ from any (initial or not) stakeholder U, $\mathcal{F}_{\text{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{C}_1, \eta_1))$ to U.

Functionality $\mathcal{G}_{\text{CLOCK}}$

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $U_p = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each session sid the clock maintains a variable τ_{sid}. For each identity $U_p := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable d_{U_p}. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})}$ (all integer variables are initially 0).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $U_p \in \mathcal{P}$ set $d_{U_p} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal—shared or local—functionality) return $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ to the requestor.

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

Functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{L}}$

The (proof-malleable) non-interactive zero-knowledge functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{L}}$ allows proving of statements in an NP language $\mathcal{L}$. It maintains a set of statement/proof pairs Π, initialized to $\varnothing$.

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*

    if $(x, w) \notin \mathcal{L}$ **then**

        return $(\mathsf{proof}, \text{sid}, x, \perp)$

    end if

    send $(\mathsf{prove}, \text{sid}, x)$ **to** $\mathcal{A}$ and **receive the reply** $(\mathsf{proof}, \text{sid}, x, \pi)$

    let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

    return $(\mathsf{proof}, \text{sid}, x, \pi)$

Proof Malleability *When receiving a message* $(\mathsf{maul}, sid, x, \pi)$ *from* $\mathcal{A}$*:*

    if $\nexists \pi' : (x, \pi') \in \Pi$ **then**

        return $(\mathsf{maul}, \text{sid}, x, \pi, \perp)$

    end if

    let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

    return $(\mathsf{maul}, \text{sid}, x, \pi, \top)$

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*

    if $(x, \pi) \notin \Pi$ **then**

        send $(\mathsf{verify}, \text{sid}, x, \pi)$ **to** $\mathcal{A}$ and **receive the reply** R

        if $R = (\mathsf{witness}, \text{sid}, x, \pi, w) \land (x, w) \in \mathcal{L}$ **then**

            let $\Pi \leftarrow \Pi \cup (x, \pi)$

        end if

end if

return (verify, sid, $x, \pi, (x, \pi) \in \Pi$)

Functionality $\mathcal{F}_{\text{FwEnc}}$

$\mathcal{F}_{\text{FwEnc}}$ is parameterized by, a security parameter κ, a set of parties $\mathcal{P}$, and a maximum delay Δ_{max}.

- **Key Generation.** Upon receiving a message (KeyGen, sid) from a party U_p, verify that $U_p \in \mathcal{P}$, and that this is the first key generation. If so, send (KeyGen, sid, U_p) to $\mathcal{A}$, and receive a value pk_p in return. Return pk_p to U_p, and initialize $\tau_p := 0$ and add U_p to the set of honest parties $\mathcal{H}$.
- **Encryption.** Upon receiving a message (Encrypt, sid, pk, τ, m) from some party U_p:
  - Check that there exists a $U_q \in \mathcal{P}$, where $pk_q = pk$ and $U_q \in \mathcal{H}$, and $\tau < \tau_q + \Delta_{\text{max}}$. If so, send (Encrypt, sid, $\tau, |m|, U_p$) to $\mathcal{A}$. Otherwise, send (DummyEncrypt, sid, pk, τ, m, U_p) to $\mathcal{A}$.
  - Receive a reply c from $\mathcal{A}$, and send (ciphertext, c) to U_p. Further, if the conditions in the previous step were satisfied, record the tuple (U_q, m, τ, c).
- **Decryption.** Upon receiving a message (Decrypt, sid, τ', c) from party $U_p \in \mathcal{P}$:
  - If $\tau' < \tau_p$, return $\perp$.
  - Else, if a tuple (U_p, m, τ', c) was recorded, return m to U_p.
  - Otherwise, send (Decrypt, sid, τ_p, c, U_p) to $\mathcal{A}$, receive a reply m, and forward m to U_p.
- **Update.** Upon receiving a message (Update, sid) from party $U_p \in \mathcal{P}$:
  1. Send (Update, sid, U_p) to $\mathcal{A}$.
  2. Update $\tau_p \leftarrow \tau_p + 1$
- **Corruptions.** Upon corruption of a party $U_p \in \mathcal{P}$, remove U_p from $\mathcal{H}$.

7.10 The Simulator

7.10.1 The Stage 1 Simulator

Procedures EXTENDLEDGERSTATE, and ADJUSTVIEW as in Ouroboros Genesis, and SIMULATESTAKING as in Ouroboros Genesis for $\mathcal{S}_1$.

Simulator $\mathcal{S}_1$ (Part 1 - Main Structure)

Overview:

- The simulator internally emulates all local UC functionalities by running the code (and keeping the state) of $\mathcal{F}_{\text{INIT}}$, $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{ENC}}$, $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$, and $\mathcal{F}^{\text{tx}}_{\text{N-MC}}$.
- The simulator mimics the execution of Ouroboros-Crypsinous for each honest party U_p (including their state and the interaction with the hybrids).
- The simulator emulates a view towards the adversary $\mathcal{A}$ in a black-box way, i.e., by internally running adversary $\mathcal{A}$ and simulating his interaction with the protocol (and hybrids) as detailed below for each

hybrid. To simplify the description, we assume $\mathcal{A}$ does not violate the requirements by the wrapper $\mathcal{W}_{\text{OG}}^{PoS}(\cdot)$ as this would imply no interaction between $\mathcal{S}_1$ (i.e., the emulated hybrids) and $\mathcal{A}$.

- For global functionalities, the simulator simply relays the messages sent from $\mathcal{A}$ to the global functionalities (and returns the generated replies). Recall that the ideal world consists of the dummy parties, the ledger functionality, and the clock.

Party sets:

- As defined in Ouroboros Genesis [BGK+18], honest parties are categorized. $\mathcal{S}_{\text{alert}}$ denote synchronized parties that are not stalled, $\mathcal{S}_{\text{syncStalled}}$ are synchronized parties that are stalled, and $\mathcal{P}_{DS}$ are de-synchronized parties.
- For each registered honest party, the simulator maintains the local state containing in particular the local chain $\mathcal{C}_{\mathsf{loc}}^{(U_p)}$, the time t_{on} it remembers when last being online, spendable coins $\mathfrak{C}$, and the log of transactions, log. For each party U_p and clock time τ, the simulator stores a flag $\mathsf{update}_{U_p,\tau}$ (initially false) to remember whether this party has updated its state already in this round. Note that an registered party is registered with all its local hybrids.
- Upon any activation, the simulator will query the current party set from the ledger, and the clock, to evaluate in which category an honest party belongs to. If a new honest party is registered to the ledger, it internally runs the initialization procedure of Ouroboros-Crypsinous.
- We assume that the simulator queries upon any activation for the sequence $\vec{\mathcal{I}}_H^T$, and the current time τ from the clock. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\text{PL}}$.

Messages from the Clock: as in Ouroboros Genesis.

Messages from the Ledger:

- Upon receiving $(\text{SUBMIT}, \mathsf{BTX})$ from $\mathcal{G}_{\text{PL}}$ where $\mathsf{BTX} := (\mathsf{tx}, \text{txid}, \tau, U_p)$, simulate running $(\text{SUBMIT}, \mathsf{BTX})$ as U_p, interacting with the simulated network $\mathcal{F}_{\text{N-MC}}$.
- Upon receiving $(\text{GENERATE}, U_p, \mathsf{tag})$ from $\mathcal{G}_{\text{PL}}$, if tag is ID, and this is the first ID query for U_p, return pk_p^{ENC}, otherwise execute GENERATE as the simulated party U_p and tag.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid})$ from $\mathcal{G}_{\text{PL}}$, extract from $\vec{\mathcal{I}}_H^T$ the party U_p that issued this query. If U_p has already completed its round-task, then ignore this request. Otherwise, execute SIMULATESTAKING(U_p, τ).

Simulator $\mathcal{S}_1$ (Part 2 - Black-Box Interaction)

Simulation of Functionality $\mathcal{F}_{\text{INIT}}$ towards $\mathcal{A}$:

- The simulator relays back and forth the communication between the (internally emulated) $\mathcal{F}_{\text{INIT}}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.
- If at time $\tau = 0$, a corrupted party $U_p \in \mathcal{S}_{\text{initStake}}$ registers via $(\mathsf{claim}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$, then input $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}_{\text{PL}}$ on behalf of U_p. Intercept the keys returned from $\mathcal{F}_{\text{INIT}}$, locally store them, and send the intercepted pk^{COIN} as the id for the coin in $\mathcal{G}_{\text{PL}}$.

Simulation of the Functionalities $\mathcal{F}_{\text{NIZK}}$ and $\mathcal{F}_{\text{FwEnc}}$ towards $\mathcal{A}$:

- The simulator relays back and forth the communication between the (internally emulated) hybrids and the adversary $\mathcal{A}$ (either direct communication, communication to $\mathcal{A}$ caused by emulating the actions of honest parties, or communication of $\mathcal{A}$ on behalf of a corrupted party). Whenever a witness is supplied for a NIZK proof, the given witness is recorded.

Simulation of the Networks $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$, and $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ as in Ouroboros Genesis, with the following modifications:

- The simulator records transactions originating from $\mathcal{A}$ or a corrupted party.
- When an adversarial transaction first enters the confirmed state, the simulator attempts to extract the witness.
- If the witness does not extract, abort.
- If the witness is successfully extracted, compute the corresponding ideal-world transaction as follows:
  - From the extracted secret keys and nonces, determine the ideal-world coins being spent. If one does not exist, abort.
  - If the public key of the "change" coin is assigned to the adversary, use it directly in the ideal transaction. If it is assigned to an honest party, generate a new adversarial ID in the ideal world for it, and record the relationship between the coins. If was not previously seen, generate it directly as an adversarial ID.
  - If the "recipient" coin public key is adversarial, use it directly as the coin ID. If it is honest, *and* the transaction's ciphertext is a correct encryption of the coin to the same honest party, use it directly as the coin ID as well. If it is otherwise honest, again generate a new adversarial ID in the ideal world, and record the relationship between the coins. If it was not previously seen, generate it directly as an adversarial ID.
  - Form an ideal-world transaction with the above coin IDs and extracted values.

7.10.2 The Stage 2 Simulator

Simulator $\mathcal{S}_2$

The Simulator $\mathcal{S}_2$ behaves like $\mathcal{S}_1$, with key differences listed below. The simulator maintains a record of simulated NIZK proofs. When asked to verify a simulated NIZK proof by the adversary through $\mathcal{F}_{\text{NIZK}}$, return $\top$ if the statement provided is the same statement recorded, otherwise return $\bot$. We define ℓ_{Coin} to be the length of coin tuples.

- Upon receiving $(\textsc{submit}, \mathsf{BTX})$ from $\mathcal{G}_{\text{PL}}$ for honest transactions, if $\mathsf{BTX} = (\textsc{public}, \textsc{transfer}) \parallel \mathsf{BTX}'$, run $\textsc{SimulateTransfer}(\mathsf{BTX}')$. Otherwise, run $\textsc{SimulateGeneric}(\mathsf{BTX})$.
- Upon receiving $(\textsc{maintain-ledger}, \text{sid})$ from $\mathcal{G}_{\text{PL}}$, extract from $\vec{\mathcal{I}}^T_H$ the party U_p that issued this query. If U_p has already completed its round-task, then ignore this request. Otherwise, execute $\textsc{SimulateStaking}(U_p, \tau, L_\tau)$, where L_τ is the leadership leakage for time τ. If this is not yet known, query $\mathcal{G}_{\text{PL}}$ with $\textsc{read}$ for it.
- Upon the adversary requesting corruption of a party U_p, corrupt the corresponding ideal-world party immediately, and run $\textsc{Corrupt}(U_p)$.

procedure SIMULATETRANSFER($(\mathsf{stx}^{\mathsf{ideal}}_{\mathsf{rcpt}}, \mathsf{stx}^{\mathsf{ideal}}_{\mathsf{chng}})$)
  if $\mathsf{stx}^{\mathsf{ideal}}_{\mathsf{rcpt}} = \bot$ **then**
    Let $cm \leftarrow \widehat{\mathsf{Comm}}(ek)$.
    Send $(\mathsf{Encrypt}, \mathrm{sid}, \tau, \ell_{\mathsf{Coin}}, U_p)$ to $\mathcal{A}$, and denote the response $\mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}}$.
  else
    Let $(pk^{\mathrm{ENC}}_q, (pk^{\mathrm{COIN}}, v)) \leftarrow \mathsf{stx}_{\mathsf{rcpt}}$
    Let $\rho \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$
    Let $(cm, r) \leftarrow \mathsf{Comm}(pk^{\mathrm{COIN}} \parallel \rho \parallel v)$
    Use $\mathcal{F}_{\mathrm{FwEnc}}$ to encrypt $(pk^{\mathrm{COIN}}, \tau, \rho, r, v)$ to pk^{ENC}_q, and denote the ciphertext $\mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}}$.
  end if
  Let $cm_2 \leftarrow \widehat{\mathsf{Comm}}(ek)$.
  Let $sn_1, sn_2 \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$
  If either ρ_1 or ρ_2 were adversarially generated, and can be read from the transaction, use them directly to compute sn_1 or sn_2 instead.
  Let root be the Merkle tree root of the current state of U_p.
  Let $\mathbf{x} \leftarrow (\{cm_3, cm_4\}, \{sn_1, sn_2\}, \mathsf{root})$
  Send $(\mathsf{Prove}, \mathbf{x}, U_p)$ to $\mathcal{A}$, denoting the response π.
  Record the pair $(\mathbf{x}, \pi)$.
  Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{cm, cm_2\}, \{sn_1, sn_2\}, \mathsf{root}, \pi)$.
  Broadcast $(\mathrm{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ as U_p.
end procedure

procedure SIMULATEGENERIC($\mathsf{tx}^{\mathsf{ideal}}$)
  Let $\mathsf{tx}^{\mathsf{real}} = \mathrm{GENERIC}$
  for $\mathsf{stx} \in \mathsf{tx}^{\mathsf{ideal}}$ in order **do**
    if $\mathsf{stx} = (pk^{\mathrm{ENC}}_i, M)$ **then**
      Send $(\mathsf{Encrypt}, \mathrm{sid}, pk^{\mathrm{ENC}}_i, \tau, M)$ to $\mathcal{F}_{\mathrm{FwEnc}}$ on behalf of U_p, and denote the response c.
      Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\bot, c)$.
    else if $\mathsf{stx} = (\mathrm{PUBLIC}, M)$ **then**
      Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\mathrm{PUBLIC}, M)$.
    else
      Send $(\mathsf{Encrypt}, \mathrm{sid}, \tau, |M|, U_p)$ to $\mathcal{A}$, and denote the response c.
      Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\bot, c)$.
    end if
  end for
  Broadcast $\mathsf{tx}^{\mathsf{real}}$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ as U_p.
end procedure

procedure SIMULATESTAKING(U_p, τ, L)
  if $U_p \notin L$ **then return**
  Let $cm \leftarrow \widehat{\mathsf{Comm}}(ek); \rho, sn \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$.
  If ρ was adversarially generated, and can be read from the transaction, use it directly to compute sn instead.
  Send $(\mathsf{Encrypt}, \mathrm{sid}, \tau, \ell_{\mathsf{Coin}}, U_p)$ to $\mathcal{A}$, and denote the response as c.
  Let B, h, ptr, ep, sl, root, η_{ep}, and $\mathsf{stx}_{\mathsf{ref}}$ be defined as in an honest staking protocol execution by U_p.
  Let $\mathbf{x} \leftarrow (\eta_{ep}, cm, sn, sl, \rho, h, ptr, \mathsf{root})$
  Send $(\mathsf{prove}, \mathbf{x}, U_p)$ to $\mathcal{A}$, denoting the response π.

Record $(\mathbf{x}, \pi)$.
Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (cm, sn, ep, sl, \rho, \pi, h, ptr)$
Let $\mathsf{tx} \leftarrow (\text{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$
Broadcast tx to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$, and (tx, B) to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ as U_p.
end procedure

procedure CORRUPT(U_p)
Corrupt U_p in the ideal protocol.
Send $(\mathsf{Read}, \mathsf{sid})$ to $\mathcal{G}_{\text{PL}}$ on behalf of U_p. From the result, compute log, depending on the receiving transactions recorded.
Register U_p with $\mathcal{F}_{\text{FwEnc}}$, and update the party's key for time τ.
Determine which leadership and transfer transactions were simulated as originating from U_p.
Disambiguate which coins won which leadership transactions.
for each unspent coin $\mathbf{c}$ belonging to U_p **do**
if $\mathbf{c}$ was created by an honest party **then**
Let $\rho_{\mathbf{c}} \xleftarrow{\$} \{0,1\}^{\kappa}$
Let τ be the time the coin creating transaction was submitted.
Let $r_{\mathbf{c}} \leftarrow \mathsf{Equiv}(ek, cm_{\mathbf{c}}, pk^{\text{COIN}}_{\mathbf{c}} \parallel \tau \parallel \rho_{\mathbf{c}} \parallel v_{\mathbf{c}})$.
else
Extract $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$ by decrypting the corresponding ciphertext.
end if
if $\mathbf{c}$ is currently visible to U_p **then**
Add $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$ to U_p's $\mathfrak{C}_{\mathsf{cnd}}$.
end if
Ensure $\mathfrak{C}_{\mathsf{free}}$, $\mathfrak{C}_{\mathsf{cnd}}$, and $\mathfrak{C}$ are consistent with a real execution, by checking which coins are confirmed, moving them to $\mathfrak{C}$, and erasing them from $\mathfrak{C}_{\mathsf{free}}$ and $\mathfrak{C}_{\mathsf{cnd}}$.
end for
end procedure

7.11 UC Specification of Ouroboros Crypsinous

Protocol Ouroboros-Crypsinous$_k(U_p, \mathrm{sid})$

Registration/Deregistration: *Intially, as in Ouroboros-Genesis, then call* Initialization-Crypsinous$(U_p,$ *sid*$, R)$*, returning the result.*

Interacting with the Ledger ***(cf. Section 7.6.4):***

Upon receiving a ledger-specific input $I \in \{(\text{SUBMIT}, \ldots), (\text{READ}, \ldots), (\text{MAINTAIN-LEDGER}, \ldots)\}$ verify first that all resources are available. **If** not all resources are available, **then** ignore the input; **else** execute one of the following steps depending on the input I:

- **If** $I = (\text{SUBMIT}, \mathrm{sid}, (\text{PUBLIC}, \text{TRANSFER}) \parallel \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitXfer}(\mathsf{tx}, \mathcal{C}_{\mathsf{loc}}, \mathsf{log})$.
- **Else if** $I = (\text{SUBMIT}, \mathrm{sid}, \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitGeneric}(\mathrm{sid})$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \mathrm{sid})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, U_p, \mathrm{sid}, k, s, R, f, \mathsf{log})$; if LedgerMaintenance halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{GENERATE}, \mathrm{sid}, \mathsf{tag})$ **then**
  - If $\mathsf{tag} = \text{COIN}$, query $\mathcal{G}_{\text{CLOCK}}$ for the current time τ. Then, sample $sk^{\text{COIN}}_{\tau} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$, and let

$sk_{i+1}^{\text{COIN}} \leftarrow \mathsf{PRF}_{sk_i}^{\mathsf{evl}}(1)$, for $i \in \{\tau+1, \ldots, \tau+R\}$. Let $\mathsf{root}_{sk}^{\text{COIN}}$ be the root of the Merkle tree over $sk_{\tau}^{\text{COIN}}, \ldots, sk_{\tau+R}^{\text{COIN}}$, and $pk^{\text{COIN}} \leftarrow \mathsf{PRF}_{\mathsf{root}_{sk}^{\text{COIN}}}^{\mathsf{pk}}(\tau)$. Insert the Merkle tree into $\mathfrak{C}_{\mathsf{free}}$, and return pk^{COIN}.

- If $\mathsf{tag} = \text{ID}$, and this is the first query for ID, send $(\mathsf{KeyGen}, \text{sid})$ to $\mathcal{F}_{\text{FwEnc}}$. Denote the response by pk^{ENC}. Record pk^{ENC}, then return it.
- Otherwise, return a uniformly sampled value from $\{0,1\}^{\kappa}$.

– **If** $I = (\text{READ}, \text{sid})$ **then** invoke protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, U_p, \text{sid}, R, f, \mathsf{log})$.

Handling external (protocol-unrelated) calls: *as in Ouroboros-Genesis.*

7.11.1 Party Initialization

Protocol Initialization-Crypsinous(U_p, sid, R)

The following steps are executed in an $(\text{MAINTAIN-LEDGER}, \text{sid})$-interruptible manner:

1: Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.
2: **if** $\tau = 0$ **then** execute the following steps in an $(\text{MAINTAIN-LEDGER}, \text{sid})$-interruptible manner:
3: Send $(\mathsf{claim}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$ to claim stake from the genesis block, receiving the response $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$, and sk^{COIN}.
4: Let $\mathfrak{C} \leftarrow \{(pk_{\mathbf{c}}^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$, and $\mathfrak{C}_{\mathsf{free}} \leftarrow \{sk^{\text{COIN}}\}$
5: Send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$.
6: Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$, and give up the activation.
7: **while** $\tau = 0$ **do**
Use the clock to update τ, ep, and sl and give up the activation.

end while
8: **else**
Send $(\mathsf{genblock_req}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$. If $\mathcal{F}_{\text{INIT}}$ signals an error then halt. Otherwise, receive from $\mathcal{F}_{\text{INIT}}$ the response $(\mathsf{genblock}, sid, \mathbf{G} = (\mathbb{C}_1, \eta_1))$.
9: Set $\mathcal{C}_{\mathsf{loc}} \leftarrow (\mathbf{G})$.
10: Send $(\text{NEW-PARTY}, \text{sid}, U_p)$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{new}}$.
11: Return $pk_{\mathbf{c}}^{\text{COIN}}$.
end if
12: Set $t_{\text{on}} \leftarrow \tau$.
13: Return $\varnothing$.

GLOBAL VARIABLES: The protocol stores the list of variables $pk^{\text{ENC}}, \tau, ep, sl, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, \mathfrak{C}_{\mathsf{free}}, t_{\text{on}}$ to make each of them accessible by all protocol parts.

7.11.2 The Staking Procedure

Protocol StakingProcedure$(k, U_p, ep, sl, \mathsf{buffer}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C})$

The following steps are executed in an $(\text{MAINTAIN-LEDGER}, \text{sid})$-interruptible manner:

1: **for** $(pk_{\mathbf{c}}^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \in \mathfrak{C}$ **do**
2: **if** $\mathbf{c}$ is not eligible for leadership **then continue**

3: Send $(\mathsf{eval}, \mathrm{sid}_{RO}, \text{NONCE} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\text{RO}}$, and denote the response μ_ρ.
4: Send $(\mathsf{eval}, \mathrm{sid}_{RO}, \text{LEAD} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\text{RO}}$, and denote the response μ_y.
5: Lookup $sk^{\text{COIN}}_{\mathbf{c},\tau}$, $\mathsf{root}_{\mathbf{c}}$, and $\tau_{\mathbf{c}}$ in $\mathfrak{C}_{\mathsf{free}}$ corresponding to $pk^{\text{COIN}}_{\mathbf{c}}$.
6: Let $\rho \leftarrow \mu_\rho^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}} \| \rho_{\mathbf{c}}}$; $y \leftarrow \mu_y^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}} \| \rho_{\mathbf{c}}}$
7: **if** $y < \mathsf{ord}(G)\phi_f(v_{\mathbf{c}})$ **then**
8: **repeat** Parse buffer' as sequence $(\mathsf{tx}_1, \ldots, \mathsf{tx}_n)$
9: **for** $i = 1$ to n **do**
10: **if** $\mathsf{ValidTx}_{\mathsf{OP}}(\mathsf{tx}_i, \vec{\mathsf{st}}||\mathsf{st}) = 1$ **then**
11: $\vec{N} \leftarrow \vec{N}||\mathsf{tx}_i$
12: Remove tx from buffer'
13: Set $\mathsf{st} \leftarrow \mathsf{blockify}_{\mathsf{OP}}(\vec{N})$
end if
end for
until $\vec{N}$ does not increase anymore
14: Set $ptr \leftarrow H(\mathsf{head}(\mathcal{C}_{\mathsf{loc}}))$; $h \leftarrow H(\mathsf{st})$
15: Set $\rho_{\mathbf{c}'} \leftarrow \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$; $sn_{\mathbf{c}} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$
16: Set $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk^{\text{COIN}} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'})$.
17: Let $\vec{\mathsf{stx}}_{\mathsf{ref}}$ be, in order, the list of leadership transactions made by U_p not in $\mathcal{C}_{\mathsf{loc}}$.
18: Let root be the root of the Merkle tree $\mathbb{C}^{\mathsf{lead}}$ in $\mathcal{C}_{\mathsf{loc}}$, after applying all transactions in $\vec{\mathsf{stx}}_{\mathsf{ref}}$. Let path be the path to $cm_{\mathbf{c}}$ in the same Merkle tree.
19: Let $\mathsf{path}_{\mathbf{c}}$ be the Merkle path to $sk^{\text{COIN}}_{\mathbf{c},\tau}$ in the secret-key Merkle tree.
20: Let $\mathbf{x} = (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, \eta_{ep}, sl, \rho, h, ptr, \mu_\rho, \mu_y, \mathsf{root})$.
21: Let $\mathbf{w} = (\mathsf{path}, \mathsf{root}_{sk^{\text{COIN}}}, \mathsf{path}_{\mathbf{c}}, \tau_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}, r_{\mathbf{c}'})$.
22: Send $(\mathsf{prove}, \mathsf{sid}, \mathbf{x}, \mathbf{w})$ to $\mathcal{F}^{\mathcal{L}_{\text{LEAD}}}_{\text{NIZK}}$, and denote the response π.
23: Let $\mathsf{tx}_{\mathsf{lead}} = (\text{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi))$.
24: Set $B \leftarrow (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st})$; $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, B$.
25: Update $\mathbf{c}$: $\mathfrak{C} \leftarrow (\mathfrak{C} \setminus \{(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}) \cup \{(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}'}, r_{\mathbf{c}'}, v_{\mathbf{c}})\}$
26: Send $(\text{MULTICAST}, \mathsf{sid}, \mathsf{tx}_{\mathsf{lead}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
27: Send $(\text{MULTICAST}, \mathsf{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
28: **break**
end if
end for
29: **while** A $(\text{CLOCK-UPDATE}, \mathsf{sid}_C)$ has not been received during the current round **do**
Give up activation. Upon next activation of this procedure, proceed from here.

end while

7.11.3 The Ledger Maintenance Procedure

Protocol LedgerMaintenance$(\ldots)$

The following steps are executed in an $(\text{MAINTAIN-LEDGER}, \mathsf{sid})$-interruptible manner:

1: Execute **FetchInformation** to receive the newest messages for this round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$, and read the flag WELCOME.

2: **if** WELCOME $= 1$ **then**
3: Send $(\text{MULTICAST}, \text{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$.
4: **for** each $\mathsf{tx} \in \mathsf{buffer}$ **do**
Send $(\text{MULTICAST}, \text{sid}, \mathsf{tx})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
end for
end if
5: **for** transaction $\mathsf{tx} \in (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$ **do**
6: **if** tx is a transfer transaction **then**
7: Attempt to decrypt each new ciphertext c by sending $(\mathsf{Decrypt}, \text{sid}, c)$ to $\mathcal{F}_{\text{FwEnc}}$. Receive the response m.
8: **if** $m = (pk^{\text{COIN}}, \tau, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \wedge cm_{\mathbf{c}} \in \mathsf{tx}$ **then**
9: **if** $\nexists sk^{\text{COIN}}_{\tau} \in \mathfrak{C}_{\mathsf{free}}$ corresponding to pk^{COIN}
10: **then continue**
11: Let $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$.
12: Let $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\mathsf{tx}, \text{RECEIVE}, (pk^{\text{COIN}}, v_{\mathbf{c}}))$.
13: **end if**
14: **else if** tx is a generic transaction **then**
15: Attempt to decrypt each subtransaction ciphertext c by sending $(\mathsf{Decrypt}, \text{sid}, c)$ to $\mathcal{F}_{\text{FwEnc}}$. Receive the response m.
16: **if** $m \neq \bot$ **then** $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\text{PLAINTEXT}, c, m)$
17: **end if**
18: **end for**
19: **for** coin $(sk^{\text{COIN}}_{\mathbf{c}}, \tau_{\mathbf{c}}) \in \mathfrak{C}_{\mathsf{free}}$ **do**
20: **if** $\exists$ a coin for pk^{COIN} in $\mathfrak{C}_{\mathsf{cnd}}$ whose transaction $\in \mathcal{C}^{\lceil k}_{\mathsf{loc}}$ **then**
21: Move such candidates to $\mathfrak{C}$.
22: **end if**
23: Erase $sk^{\text{COIN}}_{\mathbf{c},\tau}$ (and for any time before τ) from $\mathfrak{C}_{\mathsf{free}}$.
24: **end for**
25: Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.
26: Set $\mathsf{buffer} \leftarrow \mathsf{buffer}||(\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$, $t_{\text{on}} \leftarrow \tau, \mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots,$ and $\mathcal{C}_M\}$
27: Invoke Protocol $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
28: Update $\mathcal{F}_{\text{FwEnc}}$ as many times as necessary for its time to be a least $\tau - k$.
29: **if** $t_{\text{work}} < \tau$ **then**
30: Invoke protocol $\mathsf{StakingProcedure}(k, U_p, ep, sl, \mathsf{buffer}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C})$ (in a $(\text{MAINTAIN-LEDGER}, \text{sid})$-interruptible manner).
31: Set $t_{\text{work}} \leftarrow \tau$ and send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$.
end if

7.11.4 Submitting Transfer Transactions

Protocol $\mathsf{SubmitXfer}(\mathsf{tx}_{\mathsf{xfer}}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, \mathsf{log})$

1: Let $((pk^{\text{ENC}}_r, (pk^{\text{COIN}}_{\mathbf{c}_4}, v_4)), (pk^{\text{ENC}}_s, (pk^{\text{COIN}}_{\mathbf{c}_1}, v_1), (pk^{\text{COIN}}_{\mathbf{c}_2}, v_2), (pk^{\text{COIN}}_{\mathbf{c}_3}, v_3))) \leftarrow \mathsf{tx}_{\mathsf{xfer}}$.
2: **if** $pk^{\text{ENC}}_s \neq pk^{\text{ENC}}$ **or** $v_1 + v_2 \neq v_3 + v_4$ **or** $pk^{\text{COIN}}_{\mathbf{c}_3} \notin \mathfrak{C}_{\mathsf{free}}$ **then return**
3: Check $\mathfrak{C}$ for the first coin received at ID $pk^{\text{COIN}}_{\mathbf{c}_1}$, $pk^{\text{COIN}}_{\mathbf{c}_2}$. Ensure they have value v_1 and v_2 respectively, and denote their (potentially evolved) variant as $\mathbf{c}_1$ and $\mathbf{c}_2$. Ensure these are in $\mathcal{C}^{\lceil k}_{\mathsf{loc}}$.
4: As a special case, allow $pk^{\text{COIN}}_{\mathbf{c}_2} = \bot$, and $v_2 = 0$.
5: **if** these do not exist, or are not in $\mathfrak{C}$ **then return**

6: Retrieve the corresponding $(pk^{\text{COIN}}_{\mathbf{c}_i}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i})$ from $\mathfrak{C}$.
7: Lookup $sk^{\text{COIN}}_{\mathbf{c}_i}$ in $\mathfrak{C}_{\mathsf{free}}$ for $i \in \{1, 2\}$, corresponding to $pk^{\text{COIN}}_{\mathbf{c}_i}$.
8: **if** $pk^{\text{COIN}}_{\mathbf{c}_2} = \bot$, and $v_2 = 0$ **then** $sn_{\mathbf{c}_2} \leftarrow \mathsf{PRF}^{\mathsf{zdrv}}_{\mathsf{root}sk^{\text{COIN}}_{\mathbf{c}_1}}(\rho_{\mathbf{c}_1})$ and all other values for $\mathbf{c}_2$ are zeroed.
9: Sample $\rho_{\mathbf{c}_3}, \rho_{\mathbf{c}_4} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$.
10: Commit $(cm_{\mathbf{c}_i}, r_{\mathbf{c}_i}) \leftarrow \mathsf{Comm}(pk^{\text{COIN}}_{\mathbf{c}_i} \parallel v_i \parallel \rho_{\mathbf{c}_i})$, for $i \in \{3, 4\}$.
11: Let $sn_{\mathbf{c}_1} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$; $sn_{\mathbf{c}_2} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}}(\rho_{\mathbf{c}_2})$
12: Extract the state $\vec{\mathsf{st}}$ from $\mathcal{C}_{\mathsf{loc}}$.
13: Let root be the transfer Merkle tree root in $\mathcal{C}^{\lceil k}_{\mathsf{loc}}$.
14: Let path_1 and path_2 be paths to $cm_{\mathbf{c}_1}$, and $cm_{\mathbf{c}_2}$ in the same Merkle tree, respectively, or, if $pk^{\text{COIN}}_{\mathbf{c}_2} = \bot$, and $v_2 = 0$, let path_2 be empty.
15: **if** either are not found in the Merkle tree **then return**
16: Let $\mathbf{x} \leftarrow (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root})$.
17: Let $\mathbf{w} \leftarrow (\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\text{COIN}}_{\tau,\mathbf{c}_1}}, \mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\text{COIN}}_{\tau,\mathbf{c}_2}}, pk^{\text{COIN}}_{\mathbf{c}_3}, pk^{\text{COIN}}_{\mathbf{c}_4}, (\rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$.
18: Send $(\mathsf{prove}, \mathrm{sid}, \mathbf{x}, \mathbf{w})$ to $\mathcal{F}^{\mathcal{L}_{\text{XFER}}}_{\text{NIZK}}$, and receive π.
19: Send $(\mathsf{encrypt}, \mathrm{sid}, \tau, pk^{\text{ENC}}_{r}, (pk^{\text{COIN}}_{\mathbf{c}_4}, \tau, \rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_{\mathbf{c}_4}))$ to $\mathcal{F}_{\text{FwEnc}}$, and receive c_{rcpt}.
20: Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \mathsf{root}, \pi)$.
21: Let $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}} \leftarrow (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, c_{\mathsf{rcpt}})$.
22: Let $\mathsf{log} \leftarrow \mathsf{log} \setminus \{(pk^{\text{COIN}}_{\mathbf{c}_1}, v_{\mathbf{c}_1}), (pk^{\text{COIN}}_{\mathbf{c}_2}, v_{\mathbf{c}_2})\}$.
23: Erase $\mathbf{c}_{1,2}$: $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{(pk^{\text{COIN}}_{\mathbf{c}_i}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i}) \mid i \in \{1, 2\}\}$.
24: Record $\mathbf{c}_3$: $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\text{COIN}}_{\mathbf{c}_3}, \rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})\}$
25: Send $(\text{MULTICAST}, \mathrm{sid}, \mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.

7.11.5 Submitting Generic Transactions

Protocol SubmitGeneric(tx)

1: Let $\mathsf{tx}^{\mathsf{real}} = \text{GENERIC}$.
2: **for** each $\mathsf{stx} \in \mathsf{tx}$ in order **do**
3: **if** $\mathsf{stx} = (\top, M)$ **then**
4: Let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \parallel \mathsf{stx}$.
5: **else if** $\mathsf{stx} = (pk^{\text{ENC}}_{r}, M)$ **then**
6: Send $(\mathsf{Encrypt}, \mathrm{sid}, \tau, pk^{\text{ENC}}_{r}, M)$ to $\mathcal{F}_{\text{FwEnc}}$, and denote the response c.
7: Let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \parallel (\bot, c)$.
end if
end for
8: Send $(\text{MULTICAST}, \mathrm{sid}, \mathsf{tx}^{\mathsf{real}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.

7.11.6 Reading the Ledger State

Protocol ReadState$(k, \mathcal{C}_{\mathsf{loc}}, U_p, \mathrm{sid}, R, f)$

1: Execute **FetchInformation** to receive the newest messages for this round; denote the output chains by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$ (the list of transactions $(\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$ and the flag WELCOME can be ignored).
2: Invoke protocol **UpdateTime**(k, U_p, R, f) and denote the output as $\tau, ep, sl, \mathbb{S}_{ep}, \alpha^{ep}_p, T^{ep}_p$, and η_{ep}.
3: Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.

4: Set $t_{\text{on}} \leftarrow \tau, \mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$.
5: Invoke Protocol $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
6: Extract the state $\vec{\mathsf{st}}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
7: Let $\vec{\mathsf{st}}^{\mathsf{ideal}} = \epsilon$.
8: **for** each $\mathsf{tx} \in \vec{\mathsf{st}}^{\lceil k}$ in order **do**
9: **if** $\mathsf{tx} = (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}_{\mathsf{rcpt}})$ **then**
10: Let $\mathsf{stx}_{\mathsf{chng}} \leftarrow \mathsf{stx}_{\mathsf{rcpt}} \leftarrow \bot$.
11: **if** $\exists v : (\mathsf{tx}, \text{RECEIVE}, v) \in \mathsf{log}$ **then**
12: Let $\mathsf{stx}_{\mathsf{rcpt}} \leftarrow (U_p, v)$.
13: **end if**
14: Let $\vec{\mathsf{st}}^{\mathsf{ideal}} \leftarrow \vec{\mathsf{st}}^{\mathsf{ideal}} \,\|\, ((\top, \text{TRANSFER}), \mathsf{stx}_{\mathsf{rcpt}}, \mathsf{stx}_{\mathsf{chng}})$.
15: **else if** $\mathsf{tx} = (\text{GENERIC}, \mathsf{stx}_1, \ldots, \mathsf{stx}_n)$ **then**
16: Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \epsilon$.
17: **for** each subtransactions $\mathsf{stx} \in \mathsf{tx}$ in order **do**
18: **if** $\mathsf{stx} = (\top, m)$ **then**
19: Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \,\|\, (\top, m)$
20: **else if** $\mathsf{stx} = (\bot, c)$ **then**
21: **if** $\exists m : (\text{PLAINTEXT}, c, m) \in \mathsf{log}$ **then**
22: Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \,\|\, (U_p, m)$
23: **else**
24: Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \,\|\, \bot$
25: **end if**
26: **end if**
27: **end for**
28: **end if**
29: Let $\vec{\mathsf{st}}^{\mathsf{ideal}} \leftarrow \vec{\mathsf{st}}^{\mathsf{ideal}} \,\|\, (\mathsf{tx}^{\mathsf{ideal}})$.
30: **end for**
31: Output $(\text{READ}, \text{sid}, \vec{\mathsf{st}}^{\mathsf{ideal}})$.

7.12 NP Statements

Recall that we use two NIZK statements: LEAD, and XFER. XFER is very close to the statement used in Zerocash [BCG+14], while LEAD is a mixture between a Zerocash proof, and an Ouroboros Praos [DGKR18] leadership proof. We define the statements by their corresponding NP languages:
A tuple $(\mathbf{x}, \mathbf{w}) \in \mathcal{L}_{\text{LEAD}}$ iff all of the following hold:

- $\mathbf{x} = (cm_{\mathbf{c}_2}, sn_{\mathbf{c}_1}, \eta, sl, \rho, h, ptr, \mu_\rho, \mu_y, \mathsf{root})$
- $\mathbf{w} = (\mathsf{path}, \mathsf{root}_{sk^{\text{COIN}}}, \mathsf{path}_{sk^{\text{COIN}}}, \tau_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}_1}, v, r_{\mathbf{c}_2})$
- $pk^{\text{COIN}} = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}}}(\tau_{\mathbf{c}})$
- $\rho_{\mathbf{c}_2} = \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$
- $\forall i \in \{1, 2\} : \mathsf{DeComm}(cm_{\mathbf{c}_i}, pk^{\text{COIN}} \,\|\, v \,\|\, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$
- path is a valid Merkle tree path to $cm_{\mathbf{c}_1}$ in a tree with root root.
- $\mathsf{path}_{sk^{\text{COIN}}}$ is a valid path to a leaf at position $sl - \tau_{\mathbf{c}}$ in a tree with root $\mathsf{root}_{sk^{\text{COIN}}}$.
- $sn_{\mathbf{c}_1} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}}}(\rho_{\mathbf{c}_1})$

- $y = \mu_y^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \| \rho_{\mathbf{c}}}$; $\rho = \mu_\rho^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \| \rho_{\mathbf{c}}}$
- $y < \mathsf{ord}(G)\phi_f(v)$

Note that $\mathbf{x}$ of LEAD contains values sl, h, ptr that seemingly nothing is proven about. As a UC proof system is non-malleable, this makes them part of the signature of knowledge message.
A tuple $(\mathbf{x}, \mathbf{w}) \in \mathcal{L}_{\text{XFER}}$ iff all of the following hold:

- $\mathbf{x} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root})$
- $\mathbf{w} = (\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, pk^{\text{COIN}}_{\mathbf{c}_3}, pk^{\text{COIN}}_{\mathbf{c}_4}, (\rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$
- $\forall i \in \{1, 2\} : pk^{\text{COIN}}_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}}(1)$ (or, if $v_2 = 0$, this check may be skipped for $i = 2$)
- $\forall i \in \{1, ..., 4\} : \mathsf{DeComm}(cm_{\mathbf{c}_i}, pk^{\text{COIN}}_{\mathbf{c}_i} \| v_i \| \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$ (or, if $v_2 = 0$, this check may be skipped for $i = 2$)
- $v_1 + v_2 = v_3 + v_4$
- path_1 is a valid path to $cm_{\mathbf{c}_1}$ in a tree with root root.
- path_2 is a valid path to $cm_{\mathbf{c}_2}$ in a tree with root root, **or** $v_2 = 0$ and $sn_{\mathbf{c}_2} = \mathsf{PRF}^{\mathsf{zdrv}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$.
- $\mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_i}}$ is a valid path to a leaf at position τ in $\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}$, for $i \in \{1, 2\}$.
- $\forall i \in \{1, 2\} : sn_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}}(\rho_{\mathbf{c}_i})$ (or, if $v_2 = 0$, this check may be skipped for $i = 2$)

7.13 Protocol Assumptions Encoded as a Wrapper

This section includes complementary material for the main body. We sketch below the wrapper functionality that is applied to the hybrid functionalities used by Ouroboros-Crypsinous. It is a slight adaptation of the same wrapper used in Ouroboros Genesis [BGK+18], with the modification that calls to $\mathcal{F}_{\text{NIZK}}$ are restricted, not $\mathcal{F}_{\text{VRF}}$. In a nutshell, the wrapper observes the advancement of the entire system and checks whether the proportional stake of alert parties, of corrupted or de-synchronized parties, and of stalled parties are within the allowed range specified as required by our main theorems.

Functionality $\mathcal{W}^{PoS}_{\text{OC}}(\cdot)$

The wrapper functionality is parameterized by the bounds α,β on the alert and participating stake ratio, as defined in Ouroboros Genesis [BGK+18], respectively, the network delay and a value $\varepsilon > 0$ (the parameter that describes the gap between the honest and adversarial stake). The wrapper is assumed to be registered with the global clock $\mathcal{G}_{\text{CLOCK}}$ and is aware of sets of registered parties, and the set of corrupted parties.
We note that the wrapper makes checks about the distribution of stake. While this is trivial in Ouroboros Genesis, it is not immediately obvious that the wrapper knows this information in Crypsinous. The wrapper does, however, observe all network traffic, as well as all NIZK witnesses. From this, it can reconstruct exactly which party transfers stake when, and to whom. We will not describe this extraction in full detail, but note that effectively, as the wrapper is around all privacy-preserving functionalities, it has a clear view of the state. We can therefore make assertions about the stake distribution despite the addition of privacy.

General:

- Upon receiving any request I from any party U_p or from $\mathcal{A}$ (possibly on behalf of a party U_p which is

corrupted) to a wrapped hybrid functionality, record the request I together with its source and the current time.

- The wrapper keeps track of the active parties and their relative share to the stake distribution.

Restrictions on obtaining NIZK proofs:

- Upon receiving $(\mathsf{Prove}, \text{sid}, \cdot, \cdot)$ to $\mathcal{F}_{\text{NIZK}}$ from $\mathcal{A}$ on behalf of a party U_p which is corrupted or registered but de-synchronized do the following:
  1. If the fraction of alert stake relative to all active stake in this round τ so far does not satisfy the honest majority, as in Ouroboros Genesis [BGK+18] then ignore the request.
  2. Otherwise, forward the request to $\mathcal{F}_{\text{NIZK}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{NIZK}}$ returns.
- Upon receiving $(\mathsf{Prove}, \text{sid}, \cdot, \cdot)$ to $\mathcal{F}_{\text{NIZK}}$ from an alert party U_p do the following:
  1. Forward the request to $\mathcal{F}_{\text{NIZK}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{NIZK}}$ returns.
  2. If the minimal fraction (in stake) of participation (of alert parties and in total) as required by Ouroboros Genesis [BGK+18] is reached in round τ, send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ to release the clock for this round.
- Any other request is relayed to the underlying functionality (and recorded by the wrapper) and the corresponding output is given to the destination specified by the underlying functionality.

7.14 Construction NIZKs via SNARKs

We will utilise, and prove the UC-security of the lifted SNARK system presented in [KZM+15b]. Specifically, we will focus on the version presented allowing for proof-malleability, i.e. allowing the adversary to re-prove statements with a different proof object. For our purposes, this weak version is sufficient. We note that asimulation secure NIZK, as presented in [KZM+15b] is a tuple $(\mathcal{K}, \mathcal{P}, \mathcal{V}, \widehat{\mathcal{K}}, \widehat{\mathcal{P}})$. This fairly directly corresponds to a UC protocol for $\mathcal{F}_{\text{NIZK}}$, in the $\mathcal{F}^{D}_{\text{CRS}}$-hybrid world, where D is the output distribution of $\mathcal{K}(1^\lambda, \mathcal{L})$, and proving and verification are implemented as expected with the provided algorithms. We will refer to this protocol as $\mathsf{NIZK\text{-}SNARK}$. We are also guaranteed the existance of an algorithm $\mathcal{E}$ which can extract proofs, and although that may not be well-known, we note that both the environment and simulator may be assumed to have access to $\mathcal{E}$. For a security parameter κ, we are guaranteed the properties in Figures 7.1-7.4 hold. Where we use $\approx$, we mean that the statistical distance between the distributions is less than some negligible function μ of λ.

$$\forall \mathcal{L}, (x, w) \in \mathcal{L}, \mathsf{crs} \in \mathcal{K}(1^\kappa, \mathcal{L}), \pi \in \mathcal{P}(\mathsf{crs}, x, w) : \\ \Pr\left[\mathcal{V}(\mathsf{crs}, x, \pi) = \top\right] = 1$$

Figure 7.1: Perfect completeness.

Determinism We will assume that $\mathcal{V}$ and $\mathcal{E}$ are *deterministic* algorithms. If we are given a non-deterministic verification algorithm $\mathcal{V}'$, $\mathcal{E}'$, we note that we can construct deterministic algorithms $\mathcal{V}$, and $\mathcal{E}$ by fixing the

$$\forall \mathcal{L}, \mathcal{A} : \Pr\left[\mathsf{crs} \xleftarrow{\$} \mathcal{K}(1^\kappa, \mathcal{L}); \mathcal{A}^{\mathcal{P}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs}) = \top\right]$$
$$\approx \Pr\left[(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\kappa, \mathcal{L}); \mathcal{A}^{\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}},\tau,\cdot,\cdot)}(\widehat{\mathsf{crs}}) = \top\right]$$

Where $\widehat{\mathcal{P}}_1$ acts as $\widehat{\mathcal{P}}$, but aborts if it is asked to simulate a proof for $(x, w) \notin \mathcal{L}$.

Figure 7.2: Computational zero-knowledge.

$$\forall \mathcal{L}, \mathcal{A} : \Pr\begin{bmatrix} \mathsf{crs} \xleftarrow{\$} \mathcal{K}(1^\kappa, \mathcal{L}); \\ (x, \pi) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}); \\ (\mathcal{V}(\mathsf{crs}, x, \pi) = \top) \wedge (\nexists w : (x, w) \in \mathcal{L}) \end{bmatrix} \approx 0$$

Figure 7.3: Computational soundness.

$$\forall \mathcal{L}, \mathcal{A} : \Pr\begin{bmatrix} (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\kappa, \mathcal{L}); \\ (x, \pi) \xleftarrow{\$} \mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}},\tau,\cdot,\cdot)}(\widehat{\mathsf{crs}}, \mathsf{ek}); \\ w \xleftarrow{\$} \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi); \\ x \notin Q \wedge (x, w) \notin \mathcal{L} \wedge \mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top \end{bmatrix} \approx 0$$

Where Q is set set of statements x that $\mathcal{A}$ queried using oracle access to $\widehat{\mathcal{P}}$.

Figure 7.4: Simulation sound extractability.

random tape. $\mathcal{V}$ necessarily satisfies completeness and zero-knowledge, and with overwhelming probability will still satisfy soundness and simulation sound extractability. Likewise, $\mathcal{E}$ necessarily satisfies completeness, zero-knowledge, and soundness, and with overwhelming probability satisfies simulation sound extractability. If α is the fraction of random tapes for which $\mathcal{V}$ or $\mathcal{E}$ can break some property of the NIZK with a non-negligible probability of at least β, then since the sampling of the random tape and the other inputs in the security games is independant, $\mathcal{V}'$ or $\mathcal{E}'$ respectively has a probability of at least $\alpha\beta$ of breaking the same property. As β is non-negligible, and $\alpha\beta$ negligible, by assumption, α must be negligible. Therefore, with overwhelming probability, all properties hold for $\mathcal{V}$.

7.14.1 Proof of UC-Emulation

Functionality $\mathcal{F}'^{\mathcal{L}}_{\text{NIZK}}$

The protocol $\mathcal{F}''^{\mathcal{L}}_{\text{NIZK}}$ slightly idealises $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$, by ensuring that previously proved statements always verify. It is built in the $\mathcal{F}^{D}_{\text{CRS}}$ hybrid model, and keeps crs, and Π as variables. Π is initialized to $\varnothing$.

Initialization *On first activation:*

send $(\mathsf{query}, \mathsf{sid})$ **to** $\mathcal{F}^{D}_{\text{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathsf{sid}, \mathsf{crs})$

Proving *When receiving a message* $(\mathsf{prove}, \mathit{sid}, x, w)$*:*

if $(x, w) \in \mathcal{L}$ **then**
  let $\pi \leftarrow \mathcal{P}(\mathsf{crs}, x, w); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$

  return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$
else

return $(\mathsf{proof}, \mathsf{sid}, x, \bot)$
end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
if $(x, \pi) \in \Pi$ **then**

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else if $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \bot)$
end if

Lemma 19. $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ *perfectly UC-emulates* $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ *in the* $\mathcal{F}^{D}_{\mathrm{CRS}}$*-hybrid model.*

Proof. We note that (prove) queries, and CRS have identical output in $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ and $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$. For previously unseen statement/proof pairs, verification queries are also identical. For previously seen statement/proof pairs (x, π), $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ would output $\mathcal{V}(\mathsf{crs}, x, \pi)$, while $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ outputs $\top$. There are two types of "previously seen" statement/proof pairs. π may be generated by $\mathcal{P}(\mathsf{crs}, x, w)$ for some w where $(x, w) \in \mathcal{L}$. In this case, by Figure 7.1, we know that $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$, therefore the outputs are identical. Alternatively, the statement/proof pair was previously seen in verification, and $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$. Since $\mathcal{V}$ is deterministic, it will return $\top$, the same as $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$. □

Functionality $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$

The protocol $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$ is a further idealisation of $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$, which utilises simulated proofs instead of real proofs. It is built in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$ hybrid model, where D' is the output distribution of $\widehat{\mathcal{K}}(1^{\kappa}, \mathcal{L})$. It keeps $\widehat{\mathsf{crs}}$, τ, and Π as variables, where Π is initialized to $\varnothing$.

Initialization *On first activation:*
send $(\mathsf{query}, \mathsf{sid})$ **to** $\mathcal{F}^{D'}_{\mathrm{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathsf{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*
if $(x, w) \in \mathcal{L}$ **then**
let $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$

return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$
else

return $(\mathsf{proof}, \mathsf{sid}, x, \bot)$
end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
if $(x, \pi) \in \Pi$ **then**

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \bot)$
end if

Simulator $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$

let $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\lambda, \mathcal{L})$
program $\mathcal{F}^{D}_{\mathrm{CRS}}$ to return $\widehat{\mathsf{crs}}$
simulate $\mathcal{A}$

Lemma 20. *$\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ in the $\mathcal{F}^{D}_{\mathrm{CRS}}$-hybrid model UC-emulates $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$-hybrid model.*

Proof. We note that it is sufficient to prove that there exists a simulator for the dummy adversary. We will use $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$ for this purpose. We note that the difference between $\mathcal{F}^{D}_{\mathrm{CRS}}$ and $\mathcal{F}^{D'}_{\mathrm{CRS}}$ is precicely the difference in the sampled key generation parameters of Figure 7.2. Further, we note that the environment can query $\mathcal{F}^{D}_{\mathrm{CRS}}$ through the dummy adversary, which corresponds precicely to the input parameter of crs or $\widehat{\mathsf{crs}}$ in Figure 7.2.

Aside from extracting the CRS from the adversary, the environment can only make honest interactions with $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$. We note that for verification queries, these are identical, given $\mathsf{crs}/\widehat{\mathsf{crs}}$. We can therefore assume without loss of generality that the environment computes them entirely locally, issuing only $(\mathsf{prove}, \mathsf{sid}, x, w)$ queries. We note that if $(x, w) \notin \mathcal{L}$ the queries are also identical, so we assume that the adversary only makes a sequence of (prove) queries where $(x, w) \in \mathcal{L}$. Then we note that if $\mathcal{Z}$ can distinguish between $(\mathcal{D}, \mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}})$ and $(\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}, \mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}})$ with a non-negligible advantage, then $\mathcal{Z}$ is a distinguisher for Figure 7.2 with a non-negligible advantage. More precisely, we can reframe the access to $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{D}$ as access to a proving oracle and crs, and reframe access to $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$ as access to a (simulated) proving oracle and $\widehat{\mathsf{crs}}$. ☐

Functionality $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$

The protocol $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ further idealises $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$, by ensuring that not-previously seen, verifying proofs are extractable. It is built in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$ hybrid model, and keeps $\widehat{\mathsf{crs}}$, τ, ek, and Π as variables. Π is initialized to $\varnothing$.

Initialization *On first activation:*
send $(\mathsf{query}, \mathsf{sid})$ **to** $\mathcal{F}^{D'}_{\mathrm{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathsf{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*
if $(x, w) \in \mathcal{L}$ **then**
let $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$

return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$

else

return $(\mathsf{proof}, \mathsf{sid}, x, \bot)$
end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
if $(x, \pi) \in \Pi$ **then**

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
if $\exists \pi' : (x, \pi') \in \Pi$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$
else
let $w \leftarrow \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi)$
if $(x, w) \in \mathcal{L}$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
end if

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, (x, w) \in \mathcal{L})$
end if
else

return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \bot)$
end if

Lemma 21. $\mathcal{F}^{\prime\prime\mathcal{L}}_{\text{NIZK}}$ *UC-emulates* $\mathcal{F}^{\prime\prime\prime\mathcal{L}}_{\text{NIZK}}$ *in the* $\mathcal{F}^{D'}_{\text{CRS}}$*-hybrid model.*

Proof. We note that $\mathcal{F}^{\prime\prime\mathcal{L}}_{\text{NIZK}}$ and $\mathcal{F}^{\prime\prime\prime\mathcal{L}}_{\text{NIZK}}$ differ only in the verification of statement/proof pairs (x, π), where $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$, and x was not previously proved. We note that any distinguishing environment will make some number k of such queries, which must be polynomial in the security parameter.

We construct an adversary, that using $\mathcal{Z}$ breaks simulation sound extractability (Figure 7.4). $\mathcal{A}$ simulates $\mathcal{Z}$ interacting with $\mathcal{F}^{\prime\prime\prime\mathcal{L}}_{\text{NIZK}}$ (and $\mathcal{D}$), with $\mathcal{F}^{D'}_{\text{CRS}}$ programmed to return $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek})$. It records all returns w of $\mathcal{E}$, as well as the corresponding x, π inputs in a vector $\vec{e}$. If $\mathcal{Z}$ has advantage α, then with at least probability α, there must exist at least one query $\mathcal{Z}$ made where $\mathcal{F}^{\prime\prime\mathcal{L}}_{\text{NIZK}}$ and $\mathcal{F}^{\prime\prime\prime\mathcal{L}}_{\text{NIZK}}$ differed in their output. Specifically, this means that there exists a query to $\mathcal{E}$ such that the extraction failed, and $(x, w) \notin \mathcal{L}$. By the conditions that must be met before $\mathcal{E}$ is called, we also know that $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi)$, and $\nexists \pi' : (x, \pi') \in \Pi$.

$\mathcal{A}$ can test all queries made, and determine which one(s) fail extraction. It then returns (x, π) for which the extraction fails. We note that this directly breaks the extractability property given in Figure 7.4, with probability at least α. Therefore, if there is no adversary that can break simulation sound extractability except with negligible probability, there exists no environment that has a greater advantage and can distinguish $\mathcal{F}^{\prime\prime\mathcal{L}}_{\text{NIZK}}$ and $\mathcal{F}^{\prime\prime\prime\mathcal{L}}_{\text{NIZK}}$. □

Simulator $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$

Initialization *On first activation:*
send $(\mathsf{query}, \mathsf{sid})$ **to** $\mathcal{F}^{D'}_{\text{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathsf{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$
simulate $\mathcal{A}$

Simulating Proofs *On receiving a messsage* $(\mathsf{prove}, \mathit{sid}, x)$ *from* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*:*
let $Q \leftarrow Q \cup \{x\}$

return $(\mathsf{proof}, \mathsf{sid}, \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x))$

Proof Verification *On receiving a message* $(\mathsf{verify}, \mathit{sid}, x, \pi)$ *from* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*:*
if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
  if $x \in Q$ **then**
    send $(\mathsf{maul}, \mathsf{sid}, x, \pi)$ **to** $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$

    return $(\mathsf{ok}, \mathsf{sid}, x, \pi)$
  else

    return $(\mathsf{witness}, \mathsf{sid}, \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi))$
  end if
else

  return $(\mathsf{reject}, \mathsf{sid}, x, \pi)$
end if

Lemma 22. $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ *in the* $\mathcal{F}^{D'}_{\mathrm{CRS}}$*-hybrid model perfectly UC-emulates* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*.*

Proof. We will show that $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ interacting with $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ perfectly simulates $\mathcal{D}$ interacting with $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$. The environment is presented with three types of actions it can do, as before. It can verify proofs, prove statements, and query $\mathcal{F}^{D'}_{\mathrm{CRS}}$ through $\mathcal{D}$. We note that as $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ perfectly simulates the interactions between $\mathcal{D}$ and $\mathcal{Z}$, the interaction with $\mathcal{F}^{D'}_{\mathrm{CRS}}$ through $\mathcal{D}$ and through $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ is identical.

We note that when receiving a $(\mathsf{prove}, \mathsf{sid}, x, w)$ query, $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ both return $(\mathsf{proof}, \mathsf{sid}, x, \bot)$ if $(x, w) \notin \mathcal{L}$. Otherwise, $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ selects $\pi \xleftarrow{\$} \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x)$, while $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ queries $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$, which returns a value from the same distribution (note that $\widehat{\mathsf{crs}}$, and τ come from the same distribution D'. Further, both $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ add (x, π) to the set Π. In the query, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ adds x to the set Q. At each update of Π, we note that the relation $Q = \{\ x \mid (x, \pi) \in \Pi\ \}$ is preserved, and Π is equal in $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$. We will revisit this invariant at each point Π is modified. Both $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ return $(\mathsf{proof}, \mathsf{sid}, x, \pi)$.

For $(\mathsf{verify}, \mathsf{sid}, x, \pi)$ queries, we note that Π is identical in $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$, and that if $(x, \pi) \in \Pi$, both return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$. Otherwise, $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ queries $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$. If $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \bot$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ will return $(\mathsf{reject}, \mathsf{sid}, x, \pi)$, and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ will return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \bot)$, as will $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$. We note that $\exists \pi' : (x, \pi) \in \Pi \Leftrightarrow x \in Q$. Therefore, if we consider the remaining cases, where $(x, \pi) \notin \Pi$, and $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$, we have the cases that $x \in Q$ and $x \notin Q$ in both functionalities. If $x \notin Q$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ returns the witness $\mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi)$, and if $(x, w) \in \mathcal{L}$, adds (x, π) to Π. This still preserves the relation between Q and Π. Both functionalities return $(\mathsf{verify}, \mathsf{sid}, x, \pi, (x, w) \in \mathcal{L})$, and add (x, π) to Π iff $(x, w) \in \mathcal{L}$. If $x \in Q$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ sends $(\mathsf{maul}, \mathsf{sid}, x, \pi)$ to $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$, which (since $x \in Q$), permits the malleability and adds (x, π) to Π. Both will return $(\mathsf{verify}, \mathsf{sid}, x, \pi, \top)$. ☐

Theorem 23. $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ *in the* $\mathcal{F}^{D}_{\mathrm{CRS}}$*-hybrid model UC-emulates* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*.*

Proof. By the transitivity of UC-emulation. ☐

7.15 Key-Private Forward-Secure Encryption

Lifting to a UC-Protocol A kp-fs-CCA-secure key-evolving encryption scheme induces the following protocol for realizing $\mathcal{F}_{\mathrm{FwEnc}}$ in the $\mathcal{F}_{\mathrm{KeyMem}}$-hybrid model:

Protocol kp-fs-Enc

kp-fs-Enc is parameterized by Δ_{max}, κ, and N, and operates in the $\mathcal{F}_{\text{KEYMEM}}$-hybrid world, where $\mathcal{F}_{\text{KEYMEM}}$ is parameterized by Δ_{max}, and the following Update function:

function $\mathsf{Update}((sk, \tau))$

  return $(\mathsf{Upd}(sk, \tau + 1), \tau + 1)$

end function

On receiving a message (KeyGen, sid) *for the first time:*

Let $(pk, sk^0) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$.
Send $(\mathsf{Init}, \mathrm{sid}, (sk^0, 0))$ to $\mathcal{F}_{\text{KEYMEM}}$.
Erase sk^0.
Record $\tau \leftarrow 0$

return pk

On receiving a message $(\mathsf{Encrypt}, sid, pk, \tau', m)$*:*

return $\mathsf{Enc}_{pk}(\tau', m)$

On receiving a message $(\mathsf{Decrypt}, sid, \tau', c)$*:*

if $\tau' < \tau$ **then**

  return $\perp$

else

  Send $(\mathsf{Get}, \mathrm{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$, denoting the response as $(sk^\tau, \cdot)$.
  Compute $sk^{\tau'}$ from sk^τ.
  Let $m \leftarrow \mathsf{Dec}_{sk^{\tau'}}(\tau', c)$.
  Erase $sk^{\tau'}$ and sk^τ.

  return m

end if

On receiving a message (Update, sid)*:*

Record $\tau \leftarrow \tau + 1$.
Send $(\mathsf{Update}, \mathrm{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$.

The Simulator We now give the simulator for which we will show UC emulation.

Simulator $\mathcal{S}_{\text{FWENC}}$

In addition to responding to $\mathcal{F}_{\text{FWENC}}$, the simulator $\mathcal{S}_{\text{FWENC}}$ maintains a simulated $\mathcal{F}_{\text{KEYMEM}}$, through which it provides the adversary with (delayed) access to secret keys.

On initialization:

Let $(pk_{\mathsf{dummy}}, \cdot) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$
Record pk_{dummy}

On receiving a message $(\mathsf{KeyGen}, sid, U_p)$*:*

Let $(pk_p, sk_p^0) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$
Record pk_p, sk_p^0, and $\tau_p \leftarrow 0$

Simulate sending $(\mathsf{Init}, \mathrm{sid}, (sk_p^0, 0))$ to $\mathcal{F}_{\text{KEYMEM}}$ as U_p.

return pk_p

On receiving a message $(\mathsf{Encrypt}, sid, \tau, U_p, l)$*:*

Let $m \xleftarrow{\$} 0^l$
Let $c \xleftarrow{\$} \mathsf{Enc}_{pk_{\mathsf{dummy}}}(\tau, m)$

return c

On receiving a message $(\mathsf{DummyEncrypt}, sid, pk, \tau, m, U_p)$*:*

Let $c \xleftarrow{\$} \mathsf{Enc}_{pk}(\tau, m)$

return c

On receiving a message $(\mathsf{Decrypt}, sid, \tau, c, U_p)$*:*

if U_p's secret key sk_p^0 is recorded **then**
  Use Upd to derive sk_p^τ.
  Let $m \leftarrow \mathsf{Dec}_{sk_p^\tau}(\tau, c)$
  Return m
else

  return $\perp$
end if

On receiving a message $(\mathsf{Update}, sid, U_p)$*:*

Record $\tau_p \leftarrow \tau_p + 1$
Simulate sending $(\mathsf{Update}, \mathrm{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$ as U_p.

On receiving messages to $\mathcal{F}_{\text{KEYMEM}}$ *from* $\mathcal{A}$*:* Forward these messages to the simulated $\mathcal{F}_{\text{KEYMEM}}$.

UC Emulation

Theorem 24. *If the underlying key-evolving PKE scheme is kp-fs-CCA secure then* kp-fs-Enc *UC-emulates* $\mathcal{F}_{\text{FWENC}}$ *in the* $\mathcal{F}_{\text{KEYMEM}}$*-hybrid world.*

Proof. The points in which the simulator $\mathcal{S}_{\text{FWENC}}$, combined with $\mathcal{F}_{\text{FWENC}}$ can behave differently from kp-fs-Enc are in how they respond to various queries, and the internal state they maintain. kp-fs-Enc maintains a public/private key pair for each party, which the simulator selects from exactly the same distribution, and both return the public key, while storing sk_p^0. Further, both initialize τ_p to zero. As a result, for KeyGen-queries, the simulation is perfect. For update, while the simulator does not call Upd on the secret key, this is merely because the call is deferred to the point where it is used, in Decrypt. In both worlds however, τ_p is updating the same way, and matches the ideal functionality's τ_p value.

What remains is showing the correctness of encryption, decryption, and corruption queries. We will reduce this to kp-fs-CCA security, by showing that if the environment can distinguish, we can extract a kp-fs-CCA adversary with black-box access to the distinguishing environment, which wins the kp-fs-CCA game with a non-negligible advantage. In both the real and ideal worlds, the public and secret keys for $U_1, \ldots, U_n$ are sampled from $\mathsf{Gen}(1^\kappa, N)$ – with in the real-world parties holding their own keys, and in the ideal world, the simulator holding all. We note that while the dummy key pk_{dummy} exists only in the ideal world, and it's corresponding secret key is never used, we can assume it also exists in the real world, however remains entirely unused. Therefore as all (not adversarially generated) key pairs are sampled the same in both worlds, we can extract this sampling from the UC security definition – if all key pairs $(pk_1, sk_1), \ldots, (pk_n, sk_n), (pk_{\mathsf{dummy}}, sk_{\mathsf{dummy}})$ are sampled from the same distribution, and fixed in both the real and ideal executions, the real and ideal distributions are

indistinguishable with overwhelming probability. Given an environment $\mathcal{Z}$ which can distinguish between the real and ideal world with non-negligible advantage, we can therefore assume that it can distinguish between the real and ideal world, with fixed keys, with a non-negligible advantage. We use $\mathcal{Z}$ to construct an adversary $\mathcal{A}$ in kp-fs-CCA game, and prove that $\mathcal{A}$ has a non-negligible advantage. Specifically, $\mathcal{A}$ simulates running $\mathcal{Z}$ against the ideal world, with the following modifications:

- The public/secret key pairs used by the simulator are supplied by $\mathcal{A}$ by programming the random tape.
- $\mathcal{A}$ monitors all messages sent in the simulation, in particular messages to the ideal functionality from all parties.
- Since $\mathcal{A}$ does *not* hold parties secret keys, on a Decrypt query to the simulator, it posts a $\mathsf{decrypt}(\tau, c, U_p)$ query, and return the response.
- We note secret keys are only used for decryption, as well as being handed to the (UC) adversary upon corruption. When the simulator hands the keys to the (UC) adversary, the (kp-fs-CCA) adversary posts a $\mathsf{corrupt}(\tau_p + \Delta_{\max}, U_p)$ query to obtain $sk_p^{\tau_p+\Delta_{\max}}$. While $\mathcal{F}_{\text{KEYMEM}}$ at the time of corruption stores $sk_p^{\tau_p}$, by assumption it will first apply $\Delta_{\max}$ updates.
- When the ideal functionality receives an $(\mathsf{Encrypt}, \mathrm{sid}, pk_p, \tau, m)$ query, iff it does not reveal m to the simulator, $\mathcal{A}$ queries $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m))$, and returns c.

We begin by observing that this adversary does obey the rules of the kp-fs-CCA game. Specifically, the conditions for the game are as follows: a) A challenge ciphertext is not queried for decryption, and b) A party is not challenged after it has been corrupted. For a) challenge queries are performed when an Encrypt message is seen, and due to the structure of $\mathcal{F}_{\text{FwEnc}}$, the challenges will be for, at latest, the time $\tau_p + \Delta_{\max} - 1$. On corruption, the $\mathsf{corrupt}(k, U_p)$ query is make with $k = \tau_p + \Delta_{\max}$. As τ_p is monotonically increasing, and Encrypt is not called after corruption – and therefore no further challenge queries are issued – the corruption is after all challenges. For b), we note that on corruption, $\mathcal{F}_{\text{FwEnc}}$ will no longer query the simulator with Encrypt queries for this party, but only with DummyEncrypt queries. As challenge queries are only issued on Encrypt queries, this party will not longer receive challenge queries.

Next, if $b = 0$, the execution perfectly matches a random ideal world execution with $\mathcal{S}_{\text{FwEnc}}$. Specifically, if $b = 0$ the result of $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m)) = \mathsf{Enc}_{pk_{\mathsf{dummy}}}(\tau, 0^{|m|})$. Further, $\mathsf{decrypt}(\tau, c, U_p) = \mathsf{Dec}_{sk_p^\tau}(\tau, c)$, i.e. all points in which $\mathcal{A}$ intervenes in the UC execution, the execution is identical for $b = 0$.

Finally, we will argue that if $b = 1$, the statistical distance between the simulated UC execution, and the UC execution of kp-fs-Enc is negligible. Honest parties perform four operations in kp-fs-Enc: A one-time key-generation, encryption, decryption, and update. The keys are supplied in kp-fs-CCA, and sampled from the same distribution as in the protocol. τ is initialized to 0 for U_p upon key generation in both the protocol and the simulator. In both cases, pk is returned, sampled from the Gen algorithm. For encryption, regardless of whether Encrypt or DummyEncrypt is called by the functionality, as $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m)) = \mathsf{enc}_{pk_p}(\tau, m)$, the ciphertext will be sampled from $\mathsf{enc}_{pk_p}(\tau, m)$, the same as in the protocol. For decryption queries, if it lies in the past, both the protocol and functionality will return $\perp$. The functionality will, if it supplied the ciphertext itself, and the party is the intended recipient, return the corresponding plaintext. Otherwise it asks the simulator for decryption, which in turn makes a decrypt query. We note that by contrast, the protocol will *always* run $\mathsf{Dec}_{sk_p^\tau}(\tau, c)$. If a decrypt query is made, we know that – since the ciphertext was not previously challenged (at least not with the same party and time slot) – the behaviour is identical. Otherwise, we no by the correctness of the underlying key-evolving encryption scheme, that with overwhelming probability, the decryption must return the same plaintext. For update, τ_p is kept the same in the protocol and the simulated execution, by incrementing it. While the secret key is not updated in the simulated execution, this update serves only to erase information – something the simulator does not care about. □

Chapter 8

Privacy Threats Exploiting Smart Contracts and Forks

8.1 Attack to the Zero-Knowledge Property of GG-NIZK [GG17]

In this section we describe an attack with which an adversary, using a smart contract, is able to break the zero-knowledge property of GG-NIZK without corrupting any party.

In [GG17] Goyal and Goyal presented a non-interactive zero-knowledge argument of knowledge based on the assumption of the existence of a proof-of-stake blockchain. Their construction can be abstracted as an execution of a non-interactive ZAP (i.e., a non-interactive witness-indistinguishable – NIWI – proof of knowledge) where the prover proves that either a theorem is true or she possesses the majority of the stake. Without a majority of the stake, a malicious prover cannot convince the verifier with an incorrect witness. Instead the simulator will replace the honest players and thus will have a majority of the stake and this will allow it to have a witness for the ZAP. Technically, the construction relies on the prover secret sharing the witness and then encrypting the shares with the public keys corresponding to some stakeholders[1].

Our observation is that it is natural to assume that a verifier of GG-NIZK can be also a party in the blockchain protocol and therefore she can still interact with the blockchain after that she receives a proof computed by a prover, in particular an adversarial verifier can decrypt shares of the witness. More in general, the goal of this section is to bring attention to the fact that when a cryptographic protocol is constructed on top of a blockchain, the protocol designers should take into account that on top of the same blockchain there can be smart contracts run by honest players that can use their secret keys to generate valid transactions. Indeed a malicious player of the cryptographic protocol could make use of the transactions generated by honest players to invalidate some security property of the protocol. Our main idea consists of showing an adversarial verifier of GG-NIZK that produces a smart contract to decrypt shares of the witness encrypted by the prover, therefore breaking completely the privacy of the witness.

Before describing the attack, we provide a formal description of the GG-NIZK. The smart contract will include the ciphertexts corresponding to the shares of the witness encrypted with the public keys of some stakeholders. Then those stakeholders through transactions could provide verifiable decryptions of those ciphertexts therefore allowing the adversary to reconstruct the witness.

The NIZK of [GG17].

NOTATION FOR GG-NIZK

- Blockchain $\mathbf{B}$: this is the latest version blockchain which might contain unconfirmed blocks.
- Stable Blockchain $\mathbf{B}'$: this is defined as $\mathbf{B}^{\lceil \ell_1}$, which is the blockchain $\mathbf{B}$ pruned of ℓ_1 blocks (that are possibly unconfirmed blocks).

[1]Note that this approach requires that the proof-of-stake blockchain uses public keys for the stakeholders that can be used for public-key encryption.

- Parameter ℓ_2: number of last blocks taken into consideration in $\mathbf{B}'$.
- Stakeholders $\mathcal{M}$: set of public keys associated to the player that have added at least one block in the last ℓ_2 blocks of $\mathbf{B}'$. In [GG17], such public keys are crucially used for both encryption and signature.
- Chain quality parameters: ℓ_3, ℓ_4 used in the soundness proof.
- params:= $(1^{\ell_1}, 1^{\ell_2}, 1^{\ell_3}, 1^{\ell_4})$.

GG-NIZK: THE PROOF.

A proof π for theorem x is computed as follows. Let w be the witness s.t. $(x, w) \in \mathcal{R}$.

1. Secret share the witness w using a weighted secret sharing scheme, using as weights the stake of the public keys appearing in $\mathcal{M}$. Do the same with the zero-string.
   Namely, produce the following two sets[2]:
   $$\{\mathsf{sh}_{1,i}\}_{i\in\mathcal{M}} = \mathsf{Share}(w, \{s_i\}_{i\in\mathcal{M}}, \beta \cdot s_{\mathsf{total}}, s_1)$$
   $$\{\mathsf{sh}_{2,i}\}_{i\in\mathcal{M}} = \mathsf{Share}(0, \{s_i\}_{i\in\mathcal{M}}, \beta \cdot s_{\mathsf{total}}, s_2)$$
2. Encrypt each weighted share using the public key of the corresponding player. Namely for all i such that $\mathsf{PK}_i \in \mathcal{M}$, sample random strings $r_{1,i}, r_{2,i}$ and compute: $\mathsf{ctx}_{1,i} = \mathsf{Enc}(\mathsf{PK}_i, \mathsf{sh}_{1,i}, r_{1,i}) \quad \mathsf{ctx}_{2,i} = \mathsf{Enc}(\mathsf{PK}_i, \mathsf{sh}_{2,i}, r_{2,i})$.
3. Compute a non-interactive witness indistinguishable proof (NIWI) π_{niwi} for the theorem: (1) either the first set of ciphertexts are correct encryptions under the public keys in $\mathcal{M}$ of shares of the witness w or (2) (trapdoor witness) the second set of ciphertexts is a collection of correct encryptions under the public keys in $\mathcal{M}$ of shares of a valid fork of length $\ell_3 + \ell_4$.

Hence, a proof π for theorem $x \in L$ consists of the tuple:

$$\pi = (\mathbf{B}, \{\mathsf{ctx}_{1,i}, \mathsf{ctx}_{2,i}\}_{i\in\mathcal{M}}, \pi_{\mathsf{niwi}}, \mathsf{params})$$

Note that the proof π is not published on the blockchain and it is only sent to the verifier.

Security of GG-NIZK: Intuition. Zero-knowledge follows from the assumption of honest majority of stake. Under such assumption, the simulator –controlling all honest players– is able to compute a valid fork that constitutes a valid trapdoor witness for the NIWI. Even if the trapdoor witness is encrypted in $(\mathsf{ctx}_{2,i})_{\mathsf{PK}_i\in\mathcal{M}}$, a malicious verifier cannot detect that the trapdoor witness was used, since it does not control enough secret keys (associated to the public keys in $\mathcal{M}$) that would allow for collection of enough shares.

Soundness is proved by witness extraction: the extractor controls a sufficient fraction of honest secret keys (associated to the public keys in $\mathcal{M}$) and this allows the decryption of enough ciphertexts, that leads to enough shares to reconstruct the witness.

Clearly by obtaining in the future (e.g., when those keys will correspond to a reduced amount of stake) the secrets of the involved stakeholders (through adaptive corruptions or by naturally receiving the keys from honest stakeholders) the adversary would be able to decrypt those ciphertexts therefore breaking the zero knowledge property and without violating the proof-of-stake assumption. This problem imposes the assumptions/limitations of the GG-NIZK discussed previously.

A Simple Smart Contract that Breaks the ZK Property of GG-NIZK. The zero-knowledge property of GG-NIZK crucially relies on the assumption that the malicious verifier – controlling only a minority of stake– does not have enough secret keys for the public keys in $\mathcal{M}$ to be able to decrypt enough ciphertexts and thus reconstruct the witness.

Our main observation is that in order to obtain decryptions of enough ciphertexts, a malicious verifier, does not necessarily need to *own* enough of the stake/secret keys of the honest players. Instead, the malicious verifier can upload a smart contract – that we called `DecryptForMe`– where she promises a reward for a valid decryption

[2]The role of s_1, s_2 and β is not relevant for our discussion and therefore they can be ignored.

of a certain ciphertext ctx under a certain public key PK. In more details, once the malicious verifier obtains $\pi = (\mathbf{B}, \{\mathsf{ctx}_{1,i}, \mathsf{ctx}_{2,i}\}_{i \in \mathcal{M}}, \pi_{\mathsf{niwi}}, \mathsf{params})$ from an honest prover she can publish a DecryptForMe for some of the ciphertexts $\mathsf{ctx}_{1,i}$ for which she does not possess the secret key. The malicious verifier using DecryptForMe is able to collect enough shares and reconstruct the witness that is encrypted in $\{\mathsf{ctx}_{1,i}, \mathsf{ctx}_{2,i}\}_{i \in \mathcal{M}}$, thus directly invalidating the ZK property of [GG17].

In Figure 8.1 we give a more detailed description of DecryptForMe. In order to keep the smart contract simple we assume that the decryption procedure of the underlying encryption scheme gives in output a pair (m, r) where r is the randomness used to encrypt and m is the message encrypted (see for instance [BR93]). For the same reason, we also assume that (m, r) are unique (for a public key PK).

Notation (borrowed from [JKS16]).
- Ledger: the blockchain.
- $\mathsf{Ledger}[U_{p_i}]$ denotes the amount of money possessed by the secret key of party U_{p_i}.

DecryptForMe

1. Init: Upon receiving (init, $\$r, \mathsf{ctx}, \mathsf{PK}_i$) from a contractor $\mathcal{C}$:
   - Assert $\mathsf{Ledger}[\mathcal{C}] > \$r$.
   - $\mathsf{Ledger}[\mathcal{C}] := \mathsf{Ledger}[\mathcal{C}] - \$r$.
   - Set state := INIT.
2. Claim: On input (claim, v) from a player U_{p_i}:
   - Parse $v = (m, r)$.
   - If $\mathsf{ctx} = \mathsf{Enc}_{\mathsf{PK}_i}(m, r)$ then set rewards $\mathsf{Ledger}[U_p] := \mathsf{Ledger}[U_p] + \$r$.
   - Set state := CLAIMED.

Figure 8.1: Description of DecryptForMe.

Observations on the Smart Contract. We note that a player fulfilling DecryptForMe is not violating any assumption of the underlying PoS protocol or of GG-NIZK. Indeed, he is not exposing his secret key but simply providing a valid decryption of a certain ciphertext. Thus this is legitimate behavior of a honest player, she is simply executing an other application that runs on top of the blockchain.

Our smart contract is not a “bribing attack”. Bribing assumes that one is paying somebody to do something wrong/break the rules. Instead in this context an honest player is still behaving honestly and he is not breaking any rule of the underlying PoS protocol.

We also note that since the proof π is not published on the blockchain (and is not required to be), honest players could be not aware that they are helping a malicious verifier to break the security of π.

8.2 Attacking and Repairing Smart Contracts on Forking Blockchains

Typical blockchains experience delays before a transaction can be considered final. Indeed, most blockchains consists of a list of blocks that can temporary fork. In such cases, fork-resolution mechanisms decide which branch is eventually part of the list of blocks and which one is discarded, at the price of cutting off some transactions that for some time have appeared on the blockchain.

The existence of transactions that appear and then disappear from a blockchain is the source of the double-spending attack. In such attack, the adversary performs a payment through a transaction on the blockchain to receive a service off-chain. If later on the transaction related to the payment disappears from the blockchain, due to the presence of forks, then the attacker gets the money back and can spend it for something else. The crucial

point of the double-spending attack is that, while the payment transaction disappears, the obtained service is not cancelled since it is not linked to the payment transaction happening on-chain.

The double-spending problem seems to disappear when instead the service consists of another on-chain transaction that is connected to the payment transaction. Indeed, in this case, if as consequence of a fork the payment transaction disappears, then the service transaction disappears too. This chaining of transactions related to the same process can be easily implemented through smart contracts.

Since transactions take time to be confirmed in a forking blockchain, the full execution of a smart contract with multiple sequential transactions might take too long. It would thus be natural to speed up the execution of smart contracts by rushing and playing messages immediately.

We notice that forks can help an adversary to mount more subtle attacks. For example, a player could answer to some transaction A by sending another transaction B as soon as A appears on the blockchain. Obviously, in case of forks, the transaction A could appear on the blockchain in different branches, and then multiple copies of B would follow A. While at first sight, this seems to be fine, an adversary computing A can exploit its view of B in a branch of a fork to play adaptively a message different than A in another branch, invalidating some security property of the smart contract. Indeed, different transactions A1 and A2 could be played in the two branches of a fork, and transactions B1 and B2 might be required and played as answers. The honest player could become aware of the fork only after the fact. Indeed, because of a fork, transactions A1 and B1 would just disappear, and transaction A2 might appear instead. The honest player, therefore, will have to compute B2 to continue the execution of the smart contract. The fact that the adversary can play A2 adaptively after having seen B1 can produce a deviation from the expected behaviour of the smart contract, therefore compromising the appealing transparency and robustness guarantees of this technology. We want to highlights that this kind of attack will break the input independence property because in this scenario the adversary can chose its input dependently from the honest parties inputs.

We analyse a variant of the above attack focusing on the blockchain-aided MPC for parallel coin tossing. Roughly, such protocols allow a set of players to agree on a uniformly random string, and have many important applications. (For instance, they trivially imply a fair lottery.) After recalling the lottery protocol by Andrychowicz et al. [ADMM14], we show that this construction is not secure in the presence of rushing players. We then propose a new protocol that achieves security in the presence of rushing players, leveraging the power of smart contracts, but we lose the fairness guaranteed by the lottery protocol by [ADMM14].

In the Bitcoin ledger, each account is associated to a pair of keys (pk, sk), where pk is the verification key of a signature scheme—representing the address of an account—while sk is the corresponding secret key used to sign (the body of) the transactions. Each block on the ledger contains a list of transactions, and new blocks are issued by an entity called *miner*. The blockchain is maintained via a consensus mechanism based on the proof of work (PoW) [DN92]; users willing to add a transaction to the ledger forward it to all the miners, which will try to include it in the next minted block.

Due to the PoW consensus mechanism, each miner could have a different view of the ledger. The *common-prefix property* [GKL15] roughly states that all the miners have the same view of the blockchain up to a certain number k of blocks (before the last block); this guarantees long-term consistency of the transactions in the ledger. Each view of the blockchain is called a *fork*; after k blocks, with noticeable probability, one of these forks will be part of the common prefix.

We say that a transaction is *valid* if it is computed correctly (i.e., the signature is valid, the coins have not been spent already, and so on) and that it is *confirmed* if it appears in the common-prefix of all the miners (i.e., at least k blocks have passed). Each transaction Tx contains the following information:

- A set of input transactions $\mathsf{Tx}_1, \mathsf{Tx}_2, \cdots$ from which the coins needed for the actual transaction Tx are taken;
- A set of input scripts containing the input for the output scripts of $\mathsf{Tx}_1, \mathsf{Tx}_2, \cdots$;
- An output script defining in which condition Tx can be claimed;

- The number of coins taken from the redeemed transactions;
- A time lock t specifying when Tx becomes valid (i.e., a time-locked transaction won't be accepted by the miners before time t has passed).

A transaction is called *standard* if its output script contains only the signature of the account's owner (i.e., it can be redeemed by the owner by simply signing it).

The construction by [ADMM14] relies on a primitive called *time-locked commitment*. Let n denote the number of parties. Each party P_j creates $n-1$ $\mathsf{Commit}^j_{i\neq j}$ transactions containing a commitment to its lottery value. In particular, the output script of such a transaction ensures that it can be claimed either by P_j via an Open^j_i transaction exhibiting a valid opening for the commitment, or by another transaction that is signed by both P_j and P_i. Before posting these transactions on the ledger, P_j creates a time-locked transaction $\mathsf{PayDeposit}^j_i$ redeeming Commit^j_i, sends it off-chain to each $\mathsf{P}_{i\neq j}$, and finally posts all the Commit^j_i transactions on the ledger. In case P_j does not open the commitment before time t, then each recipient of a $\mathsf{PayDeposit}^j_i$ transaction can sign it and post it on the ledger; since time t has passed, the miners will now accept the transaction as a valid transaction redeeming Commit^j_i.

In more detail, the protocol works as follows.

Deposit phase: Each player P_j computes a commitment $y_j = \mathsf{Hash}(x_j||\delta_j)$, where δ_j is some randomness, sends off-chain the $\mathsf{PayDeposit}^j_i$ transactions (with time-lock t) to each $\mathsf{P}_{i\neq j}$, and posts the Commit^j_i transactions on the ledger.

Betting phase: P_j bets one coin in the form of a standard transaction $\mathsf{PutMoney}_j$ (redeeming a previous transaction held by P_j, and with P_j's signature as output script). All the players agree and sign off-chain a $\mathsf{Compute}$ transaction taking as input all the $(\mathsf{PutMoney}_j)_{j\in[n]}$ transactions, and then the last player that receives the $\mathsf{Compute}$ transaction posts it on the ledger. In order to claim this transaction, a player $\mathsf{P}_{w'}$ must exhibit the openings of the commitments of all participants: The script checks that the openings are valid, computes the index of the winner w (as a function of the values $x_1,\ldots,x_n$), and checks that $w' = w$ (i.e., the only participant that can claim the $\mathsf{Compute}$ transaction is the winner of the lottery).

Compensation phase: After time t, in case some player P_j did not send all of its $\{\mathsf{Open}^j_i\}_{i\in[n],i\neq j}$ transactions, all the other players $\mathsf{P}_{i\neq j}$ can post the $\mathsf{PayDeposit}^j_i$ transaction on the ledger, thus obtaining at least a certain number of coins as compensation.

The idea behind our attack is that, in the presence of rushing players, the protocol's messages can end-up on unconfirmed blocks. By looking at different forks, an attacker can try to change an old unconfirmed transaction by re-posting it, with the hope that it will end-up on a different fork and become part of the common prefix.

The construction described in the previous section relies on the assumption that the players are non-rushing. In particular, each player P_j should wait to post its $\mathsf{PutMoney}^j_i$ transaction only after all the Commit^j_i transactions are confirmed on the ledger, in such a way that all players are aligned on the same fork (and so the miners have the $\{\mathsf{Commit}^j_i\}_{i\in[n],j\neq i}$ transactions in their common prefix).

In the case of rushing players, when a fork occurs, an attacker can take advantage of openings of the other parties played in a faster branch in order to bias the result of the lottery on a slower branch. If eventually the slower branch remains permanently in the blockchain, then clearly the attack is successful.

For concreteness, let us focus on Blum's coin tossing, in which the winner is defined to be $w = x_1 + \ldots + x_n \bmod n+1$. Consider the following scenario:

- The (rushing) players $\mathsf{P}_1,\ldots,\mathsf{P}_n$ run a full instance of the lottery protocol; note that this requires at least 3 blocks.
- The attacker P_n hopes to see a fork containing all the $\{\mathsf{Commit}^i_j\}$ transactions of the other $n-1$ players.

- Since the attacker P_n now knows the opening $x_1, \ldots, x_{n-1}$ of the other participants, it can post a new set of $\{\mathsf{Commit}'^{i}_{n}\}_{i \in [n], i \neq n}$ transactions containing a commitment to a value x'_n such that $x_1 + \ldots + x_{n-1} + x'_n \mod n+1 = n$.

- In case the new set of transactions ends up on a different fork which is finally included in the common prefix, P_n wins the lottery.

In the above scenario, the attacker can freely bias the result of the lottery. To understand the feasibility of this attack, let's consider the implementation of this protocol in Ethereum. The number that usually are taken to consider a block final on the blockchain. Since the execution of full instance of the lottery protocol requires at least 3 blocks, the attacker has 9 blocks to publish a different commitment on a different branch of the blockchain.

We now present a smart contract that can be implemented in Ethereum that allows to run a parallel coin-tossing protocol with rushing players. The pseudocode of the smart contract proposed in this chapter is shown in Fig. 8.3.

The main challenge is that the protocol must prevent that an adversary chooses adaptively in a branch of a fork its contribution to the coin tossing after having seen the contributions of others in other branches. We tackle this problem by requiring that honest players compute their contributions evaluating a verifiable unpredictable function (VUF) [MRV99] on input the public keys of all players. Notice that if the adversary sees some evaluations of the VUF in a branch, and changes its public key in another branch, then the VUF evaluations of the honest players on this other branch are unpredictable, and thus the adversary will not manage to control the final output. Moreover, changing the public key is the only possibility for the adversary because the VUF has a uniqueness property, and thus, once the public key is selected, the adversary can play a single message that is accepted as correct by the honest players.

Informally, the protocol works as follows:

- A contract is published on the blockchain with the unique identifier sid.

- Every player that wants to play in the protocol generates the public and private keys for the VUF as $(pk, sk) \leftarrow \$\mathsf{Gen}(1^\lambda)$, and shares pk with all the protocol participants publishing it in on the contract.

- For all $i \in [n]$, using the VUF, P_i evaluates $y_i = \mathsf{Eval}(sk_i, \mathsf{Hash}(pk_1||...||pk_n||\mathrm{sid}))$ and its proof ϕ_i, and publishes (y_i, ϕ_i) on the smart contract.

- The smart contract checks for all $i \in [n]$ that y_i is obtained using a VUF with input $\mathsf{Hash}(pk_1||...||pk_n||\mathrm{sid})$, where pk_i is the address of the i-th player. Then the smart contract defines the output as $\mathsf{Hash}(y_1||...||y_n)$.

Intuitively, the adversary can not bias the output of the protocol because, as long as the input of Hash is unpredictable, the final output will be random, since we model Hash as a random oracle.

Moreover, unpredictability of the input to Hash comes from the unpredictability of the output of a VUF of at least one honest player for any sequence of public keys, and from the uniqueness of the output of the adversary once its public keys are established.

Our solution can be implemented on Ethereum. Ethereum transactions include two types of accounts: *externally-owned accounts* (EOAs), controlled by the private keys of the users, and *contract accounts* (CAs), controlled directly by their code [Woo19]. Both types of accounts have a balance in ETH (also denoted Ξ). The blockchain tracks the state of every account in its blocks. The transactions posted by an EOA consist of a destination address, a signature s, the amount of *wei* (subunits of Ξ), and an optional data field representing the inputs of a contract (which we consider as messages to a contract).

Each time an EOA creates a transaction, it is replicated to all the miners of the blockchain. If such a transaction contains inputs to some contract, it triggers the specified contract via a function call: each miner, when receiving a new transaction during the replication process, checks for the triggered contract on the blockchain, and starts to run it using its current state state, and the inputs specified in the transaction. Contracts can send messages (i.e., transactions) to the EOAs or other contracts, which means that when a miner runs the code of

a contract with some prescribed input, the output contains messages (i.e., transactions) directed to some other entity. We can model both EOAs transactions and messages from/to a contract as different types of transactions. In particular, a block might contain the following elements.

Message transactions: Represented as a tuple $\mathsf{Tx}_{\mathsf{msg}} = (\mathcal{T}, s, pk, \vec{x}, \mathtt{data})$, where $\mathcal{T}$ is the set of transactions from which $\mathsf{Tx}_{\mathsf{msg}}$ is redeeming,[3] s is the signature of the owner, pk is the public key of the receiver (either a contract or an EOA), $\vec{x}$ is the vector of inputs to the contracts (in case the receiver is a contract), and `data` represents some extra data.

Contract transactions: Represented as a tuple: $\mathsf{Tx}_{\mathsf{cnt}} = (\mathcal{T}, s, \mathtt{Balance}, \Psi)$, where `Balance` is the initial balance of the contract, Ψ is the code of the contract, and $\mathcal{T}$, s are the same as in message transactions. Each time a new transaction triggers some contract, the miners check that it is valid (e.g., that the inputs are correctly formed, and that the balance is enough). The program Ψ takes as input the current state of the contract state and a vector $\vec{x}$, and outputs a sequence of transactions $(\mathsf{Tx}_1, \mathsf{Tx}_2, \ldots)$ to possibly different recipients and a new state state'; when a miner includes a message transaction Tx_i in a new block, it also updates the current state of the contract.

Each contract has associated a sequence of states $(\mathsf{state}_1, \mathsf{state}_2, \cdots)$, where state_i is the i-th state of the contract. Typically, the state includes the global variables of the contract plus all messages sent/received from/by the contract itself. Note that, in case of forks, different miners can see diverging states for the same contract, i.e., $\mathsf{state} = (\mathsf{state}_1, \ldots, \mathsf{state}_i, \mathsf{state}_{i+1}, \cdots)$ and $\mathsf{state}' = (\mathsf{state}_1, \ldots, \mathsf{state}_i, \mathsf{state}'_{i+1}, \cdots)$.

When interacting with a smart contract, a player sends message transactions without waiting the minimum number of blocks that guarantee all the miners have a consistent view of the contract's state. This, in particular, means that if we describe the lottery protocol of [ADMM14] using a single contract, the attack previously defined still works, since the protocol's messages sent by the committer to the contract would appear in different blocks of the blockchain, which makes the protocol insecure in the presence of rushing players.

We are now ready to describe our protocol for parallel coin tossing. Our construction follows the steps described Fig. 8.2.

Our parallel coin tossing protocol has two limitations. The first one is related to the fact that the protocol 8.2 does not guarantee fairness with penalties. The second one is mainly a consequence of the first one since in the absence of a penalty in case of misbehaviour an attacker has an advantage in changing its messages of the protocol executed in another branch of the blockchain. Indeed, by changing its messages the adversary unfairly increases the probability to obtain an output that it likes more.

To overcome the above two limitations we propose a branch-dependent execution of the protocol. In this case, when the execution of the protocol is repeated in another branch, even when the very same messages are played by honest players, the output of the protocol would change and be (computationally) independent of the value obtainable in the original branch. Thus the adversary has no advantage in changing its messages when playing in a different branches.

[3] The miners, in order to validate the transaction, must check that the signer is also the owner of the redeemed transactions, and that those have a high enough balance.

Parallel Coin Tossing protocol π

Running with hard-wired parameters $\bar{q}, t_1, t_2$. Let $(\mathsf{Gen}, \mathsf{Eval}, \textsc{Prove}, \mathsf{Ver})$ be a VUF with range $\mathcal{R} \equiv \{0,1\}^\ell$, and $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function. After the smart contract of Fig. 8.3 has been created by one of the players, the protocol continues as follows:

- Any willing player deposits $d \geq \bar{q}$ coins and specifies a public key pk by triggering the deposit function, where $(pk, sk) \leftarrow \$\mathsf{Gen}(1^\lambda)$.
- After time t_1, let n be the number of participants and denote by P_i the player that made the i-th deposit (specifying public key pk_i). Each player P_i, in an arbitrary order, executes the steps below:
  - Compute $y_i = \mathsf{Eval}(sk_i, x)$, $\phi_i = \textsc{Prove}(sk_i, x, y_i)$, where $x = \mathsf{Hash}(pk_1||\ldots||pk_n||\text{sid})$ and the values $(pk_j)_{j \neq i}$ and sid are retrieved from the state of the contract.
  - If the flag redeemPhase contained in the state of the contract is set to true, trigger the claim function with inputs (y_i, ϕ_i).
- After time t_2, upon receiving a compensation from the contract (meaning that some party aborted) stop the execution. Else, retrieve $y_1, \ldots, y_n$ from the contract and output $\mathsf{Hash}(y_1||\cdots||y_n)$.

Figure 8.2: Coin-tossing protocol.

```
pragma solidity ^0.4.0;

contract ParallelCoinTossing {
  struct Player {
    bool isPlaying;
    bool hasClaimed;
    string pk;
    uint d; //Player's deposit
    uint c; //Player's claim
    }
  address[] playersAddr;
  mapping(address => Player) players;
  uint sid;

  //flags
  bool claimPhase = false; //true if the claimPhase starts

  //common input of the VUF
  uint VUFmessage;

  function beginCoinTossing() {
    sid = ...; //generate a session id
  }
  function deposit(string pubKey) public payable {
    require (msg.sender.balance >= minDep && msg.value >= minDep && players[msg.sender].d == 0 && now < time1);
    playersAddr.push(msg.sender); //add the public key of the current sender
    Player p = players[msg.sender];
    p.isPlaying = true;
    p.pk = pubKey;
    p.d = msg.value;  //msg.value is the deposit value of the player
    }
  function claim(uint rand, uint proof) public {
    require (claimPhase && now < time2 && players[msg.sender].isPlaying && !players[msg.sender].hasClaimed && VUFCheck(VUFmessage,rand,
        proof,players[msg.sender].pk));
    Player p = players[msg.sender];
    p.c = rand;
    p.hasClaimed = true;
    msg.sender.transfer(p.d);
}

  //automatic check functions run after a certain time
  function checkDeposit() public {
    require (!claimPhase && now >= time1);
    uint n = playersAddr.length;
    VUFMessage = sha3(players]playersAddr[0]].pk||...||players[playersAddr[n-1]].pk||sid);
    claimPhase = true;
    }
  function checkClaimed() public { //if the second timestamp has passed and some player didn't redeem, penalize the players
    require(claimPhase && now >= time2);
    for (uint i = 0; i < playersAddr.length; i++) {
      address pAddr = playersAddr[i];
      if (players[pAddr].c == 0)
        penalize(pAddr);
      }
    }

  //local functions
  function penalize(address penalized) private {  //penalize dishonest players by sending the compensation
    for (uint j = 0; j < playersAddr.length; i++) {
      address pAddr = playersAddr[i];
      uint n = playersAddr.length;
      if (pAddr != penalized) pAddr.transfer(minDep/n);
      }
    }
}
```

Figure 8.3: Pseudocode implementation of our smart contract for realizing parallel coin tossing. For simplicity, we omit an explicit definition of the `VRFCheck` function and of the concatenation function in the computation of `VRFMessage`.

Chapter 9

Verifiable MPC with Blockchain

9.1 Introduction

Properly embedding secure computation in its real-world context is a common challenge with secure multiparty computation (MPC). This basic issue can be illustrated with Yao's famous "Millionaires' Problem". Yao proposed a protocol that allows two parties to decide which of them holds the largest input value (e.g., their fortunes). The protocol only reveals which millionaire is the richest. However, the protocol also assumes that both parties enter their inputs honestly. Nothing stops the parties from using false input values.

Typically, in an MPC setting, the parties that provide input are also part of the computation. However, one can also imagine an outsourcing scenario where the input is external to the computation. In this case, we would like to verify the correctness of the result even if all computation parties are corrupt. This chapter explains the notion of *verifiable multiparty computation* that addresses this scenario explicitly.

Outsourcing computation is increasingly relevant in the blockchain context. For example, imagine a scenario where players in an auction post bids to a smart contract that outsources the secure computation of the result to a separate network of MPC parties. The MPC parties, who are external to the blockchain, download the private inputs from the blockchain, compute the auction result and post it back to the chain. With verifiable MPC, this computation result is accompanied with a cryptographic proof of correctness, such that blockchain users can verify the validity of the outsourced computation. We will argue in this chapter that a blockchain and verifiable MPC are a strong combination that can ensure verifiability of inputs and outputs as well as privacy of inputs and computation (and outputs, if required by the use-case).

To motivate this with a concrete example: A complete solution to the millionaires problem would guarantee that the input values accurately reflect the millionaires' fortunes and the outcome of the comparison is verifiable. Therefore, we upgrade the "Millionaires Problem" to the "Billionaires' Problem", where the parties running the protocol should be able to verify that the inputs and outputs are valid. Validity of inputs may for instance be checked against the information in a cryptographic commitment that is signed by the tax authority.

We would like to state the Billionaires' Problem in an even more interesting setting. We do not only require that the billionaires can decide among themselves who is richer, but also that any interested party should be able to verify who is richer. Moreover, any such observer should also be convinced of the outcome of the ranking without actively taking part in the secure computation.

> **Definition** The World's Billionaires' Problem.
> *Given everybody's committed tax return statements on a blockchain, produce a list of the top 400 billionaires world wide including a proof of correctness, without leaking any information of the people who do not appear on the list.*

The net-worth for the 400 richest people may be revealed, but that's not necessary.[1]

[1] For a real-world motivation of the Billionaires' problem we refer to the following article on CNBC.com: 'Forbes says Commerce Secretary Wilbur Ross lied about being a billionaire' (https://www.cnbc.com/2017/11/07/forbes-commerce-secretary-wilbur-ross-lied-about-being-a-billionaire.html).

Solving the Billionaires' problem requires the scheme to certify inputs and outputs of an MPC protocol. This may require a setup stage, depending on the choice of building blocks. In our example, a blockchain is used to handle necessary information for the proofs of correctness or "computation signatures". These computation signatures are constructed by the MPC parties as a zero-knowledge proof. A sketch of a solution to the World's Billionaires' problem is presented in Section 9.4.2. We start with an explanation of the preliminaries.

The following sections cover the following topics:

- Section 9.2 explains the notions of outsourcing and privacy-preserving, publicly verifiable outsourcing;
- Section 9.3 presents recent results in verifiable multiparty computation;
- Section 9.4 introduces the interplay between blockchain and verifiable MPC and presents a solution sketch to our Billionaires' Problem;
- Section 9.5 finishes with a conclusion.

9.2 Preliminaries

This section aims to provide an accessible introduction to privacy-preserving and publicly verifiable computation. It then discusses MPC, verifiable computation with zero-knowledge (ZK) and blockchain in more detail. These notions are key building blocks for a proposed verifiable MPC scheme based on a blockchain. MPC, zero-knowledge and blockchain are explained in Sections 9.2.2, 9.2.3 and 9.2.4 respectively. We start with an explanation of privacy-preserving and publicly verifiable outsourcing.

9.2.1 The Notion of Outsourcing

Privacy-preserving outsourcing *to a single worker* typically relies on expensive primitives, such as fully homomorphic encryption [GGP10, FGP14] and functional encryption [GKP$^+$13].[2] The practical alternative to these primitives is *privacy-preserving computation by multiple workers*, i.e., MPC. MPC protocols can guarantee correctness up to all-but-one corrupt workers [DPSZ11]. This is satisfactory when an (honest) client participates as a worker in the MPC protocol, but not when the client is an outsider to the protocol, i.e. in the case of *outsourcing*.

Publicly verifiable outsourcing to a single worker saw a breakthrough in efficiency with the publication of the Pinocchio protocol [PHGR13], based on the work of [GGPR13, Gro10]. The achievement of "verifable computation" is to allow a client to outsource a computation to a worker and cryptographically verify the result with less effort than performing the computation itself. The notion of *public verification* refers to the property that anyone, particularly someone that does not participate in the protocol, is able to verify correctness of the computation. Privacy is not a property in these protocols, however.

Privacy-preserving and publicly verifiable outsourcing to multiple workers, i.e. where the computation is private and efficiently verifiable by a client external to the MPC protocol, is an area of active research as well [BDO14, SV15, SVdV16, Vee17]. These protocols guarantee correctness even if all workers are corrupt. (Note: Privacy cannot be guaranteed when all parties are corrupt. This is inherent to MPC.)

The Trinocchio and Geppetri protocols are relevant recent work to achieve *privacy-preserving, publicly verifiable outsourcing* by integrating ZK proofs with MPC [SVdV16, Vee17]. These protocols rely on a *bulletin board* for *authenticated broadcast messages*. In this chapter we formalize these protocols in case the bulletin board is instantiated by a *blockchain* and use it to present a solution sketch to the Billionaires' Problem.

Hence, the cryptographic building blocks underpinning our target scheme are:

- Secure multiparty computation;

[2] Please see [ABC$^+$15] for an accessible overview of Fully Homomorphic Encryption, Functional Encryption and the state-of-the-art per 2015.

- Verifiable computation with zero-knowledge;
- Blockchain.

The following Sections (9.2.2 to 9.2.4) summarize the preliminaries for each bulding block, referring to earlier PRIViLEDGE deliverables where relevant.

9.2.2 Secure Multiparty Computation

MPC enables multiple parties to correctly and privately evaluate a function of private inputs. Typical applications of MPC include (electronic) voting, auctions, linear programming, linear regression and decision tree learning. Typically two settings are distinguished in which correctness and privacy properties depend on the number of malicious workers:

- Honest majority of workers (requiring ≥ 3 workers);
- Dishonest majority (requiring ≥ 2 workers).

In the case of an honest majority, MPC protocols can guarantee *fairness* and *information theoretic security*, i.e. perfect privacy. (First feasibility results on information theoretic MPC are from [CCD88, BGW88].) The fairness property formalizes the following: if one party receives the results of a computation, then all parties receive that same result. Hence, fairness is a key attribute of correctness. Information theoretic security in this context means that even a computationally unbounded adversary cannot learn anything (beyond what can be deduced from the output) from participating in the protocol.

In traditional MPC protocols, the security analysis considers malicious participants in the protocol (i.e. workers) and implicitly assumes that at least one participant is honest (i.e., the client). However, if we change our perspective to a client that is external to the MPC computation (i.e., in the outsourcing scenario), we can introduce two relevant scenarios:

- Clients or participants provide false inputs;
- All participants (workers) are corrupt.

As we illustrated with Yao's Millionaires' Problem [Yao82, Yao86] in Section 9.1, MPC protocols cannot detect when the millionaires provide false inputs and cannot guarantee correctness and privacy in case that all workers are corrupt. In subsequent sections, we present a scheme that ensures verified inputs (through cryptographic commitments) and publicly verifiable correctness of outputs.

Our scheme builds on Shamir secret-sharing [Sha79] in the honest-majority setting (i.e., with three or more parties). Canonical results in this multiparty setting are by [GMW87,BGW88,CCD88]. Throughout this chapter, we denote a secret share by $[a]$ for a finite field element $a \in \mathbb{F}$. We refer to PRIViLEDGE deliverable D2.1 [Kri18], Chapter 5 "Secure Computation", for a more elaborate treatment of MPC and subtopics.

9.2.3 Verifiable Computation with Zero-Knowledge

Verifiable Computation is a form of computation outsourcing that enables a client, or a party external to a computation, (the "verifier") to verify computations performed by untrusted worker(s) (the "prover(s)"). When workers participate in Verifiable Computation, they are required to provide a proof of correctness together with the result of the outsourced computation. Furthermore, the "zero-knowledge" property enables scenarios where the worker can include a private input to the verifiable computation and ensures that the client (or verifier) learns nothing about the private information of the worker beyond the output of the computation. For example, it enables a client to anonymously authenticate workers or aggregate sensitive data of multiple input parties.

Our verifiable MPC protocol takes the Pinocchio system [PHGR13] as a starting point. Pinocchio is widely cited, as it is regarded one of the first practical setups for verifiable computation. However, we plan to adopt

other constructions of zero-knowledge arguments of knowledge (e.g. Bulletproofs [BBB+18], Groth's SNARKs [Gro16]) as well. Pinocchio achieves efficient verifiable computation by combining a computational model called "quadratic programs" [GGPR13] and "pairing-based non-interactive arguments" [Gro10] with own refinements and engineering. We will elaborate on these concepts below.

At a high level of abstraction, the Pinocchio verifiable computation system works as follows:

- Client outsources the computation of a function F over a field $\mathbb{F}$. F is represented by a circuit C of size m, d multiplication gates and N input/output gates. For size m, $m = N + d$ holds.
- Circuit C is represented as a "quadratic program" Q over $\mathbb{F}$. The quadratic program Q is (completely) represented by three sets of $m + 1$ polynomials over $\mathbb{F}$, $\{v_k\}, \{w_k\}, \{y_k\}$, for $k \in \{0, ..., m\}$.
- A worker taking on the outsourced computation of F, evaluates circuit C, which yields the circuit wire values denoted by vector $c = (c_1, ..., c_m) \in \mathbb{F}^m$. Note that vector c is the *witness* in the *zero-knowledge proof*.
- A trusted party generates the key material, which consists of a secret trapdoor and a public Evaluation Key EK_F and Verification Key VK_F. The public EK_F, VK_F and proof π consist of elements in an (elliptic curve) group that hides witness information in the exponent. The trapdoor contains a secret point $s \in \mathbb{F}$.
- To prove correct evaluation of F, the worker constructs a polynomial p and its divisor h, where p is dependent on vector c and polynomials of quadratic program Q as follows: $p = v \cdot w - y$ for polynomials v, w, y defined as $v = \sum_i c_i \cdot v_i(x)$ (similar definitions for w and y).
- With p and h, worker can construct all necessary elements of proof (signature) π by using the public evaluation key EK_F; π contains terms necessary to construct polynomial p evaluated in secret s.
- A cheating prover trying to construct an evaluation of p in secret point s without knowing c is unsuccessful due to the Schwartz-Zippel Lemma.
- The client (or verifier) uses the public verification key VK_F to efficiently validate proof π; efficiency comes from the fact that the proof contains a small number of (elliptic curve) group elements and verification requires a small number of (pairing) operations.

The central idea of verification is to prove satisfaction to all quadratic program equations. Prover demonstrates that evaluations of the quadratic program polynomials Q in a secret point s, "in the exponent", correspond to the verification key elements.

The Pinocchio protocol consists of three polynomial-time algorithms:

- $\textit{Generate Function Keys}(F; 1^\lambda) \rightarrow (EK_F; VK_F)$
  - Trusted party creates public evaluation and verification keys for F;
  - F is represented by circuit of size m;
- $\textit{Compute and Prove}(EK_F; u) \rightarrow (y; \pi_y)$;
  - Worker evaluates circuit for $F(u)$ to obtain $y \leftarrow F(u)$ and wire values $\{c_k\}_{k \in 1,...,m}$;
  - With circuit wires, worker computes proof π_y (bilinear group elements);
- $\textit{Verify}(VK_F, u, y, \pi_y) \rightarrow \{0, 1\}$
  - Verifier uses pairing operations to efficiently verify proof;
  - Checks if π_y contains encodings ('in the exponent') of terms of polynomials v, w, y, evaluated in secret point s, which construct $p(s)$.

In more detail, the $Verify$ algorithm actually consists of five steps: the three steps "V", "W" and "Y" check that the proof is a proof of knowledge of the witness (i.e. correct wire values for left input (V), right input (W) and multiplication wires (Y)), "Z" checks that the same witness was used for steps V, W, Y, and step "H" checks that $p = h \cdot t$ holds (for a specific "target" polynomial t.)

One important aspect of Pinocchio is that it requires a setup step where a trusted party generates a common reference string (CRS), which consists of several random elements in the field $\mathbb{F}$. The trusted party is required to use these random elements from the CRS to generate the aforementioned public key material $(EK_F; VK_F)$. The trusted party is then required to delete these random elements, because an adversary could generate fake proofs if it could obtain these elements. (Note: The initial concept of using a "reference string" for non-interactive zero-knowledge is from [BSMP91].)

We refer to [PHGR13] for the complete treatment of Pinocchio, and to PRIViLEDGE deliverable D2.1 [Kri18], chapter 6 "Zero-Knowledge", for the broader topic of ZK.

9.2.4 Blockchain

Our verifiable MPC protocol could use a *blockchain* to implement the "authenticated broadcast channel functionality" of a *bulletin board*. We follow the formal definition of blockchain as a "public ledger" by [PSS17,GKL15]. In their definition, the public ledger serves as an immutable "bulletin board", where anyone can post and read a message. Garay et al. [GKL15] define two key properties for public ledgers:

- *Liveness*: If all honest players want to add a message, it should eventually appear on the ledger;
- *Persistence*: A message added to the public ledger cannot be removed.

Authentication to post to the public ledger happens via public key cryptography and verification of signatures. Typically, when a new message is posted, all honest participants validate its signature. The details of message or transaction verification in actual blockchains is explained further in Chapter 4 of PRIViLEDGE deliverable D3.1 [KC18].

We also briefly introduce the concept of a *smart contract*. A smart contract is a (public) function with a public state, hosted on a blockchain. The smart contract can be interacted with by sending a transaction to its contract interface. The smart contract computation result can be retrieved by monitoring the smart contract's public state. We suggest the HAWK paper [KMS+15] for an elaborate treatment of smart contracts and specifically smart contracts with privacy features.

We also refer to PRIViLEDGE deliverable D3.1 [KC18] and Chapters 6 and 7 in this document for an in-depth treatment of general and specific blockchains. Furtermore, we refer to PRIViLEDGE deliverable D2.2 [SV18], chapter 6 "Bulletin Boards and BPK Model" for an elaborate treatment of bulletin boards.

9.3 Recent Related Work in Verifiable Multiparty Computation

Building on the preliminaries of verifiable computation from Section 9.2.3, we now provide an introduction of verifiable computation in the MPC setting. Specifically, we introduce the Trinocchio protocol by Schoenmakers, Veeningen and De Vreede [SVdV16] and its successor Geppetri [Vee17]. Both have valuable properties to address the Billionaires's problem.

9.3.1 Trinocchio: Verifiable Computation on Private Inputs

The main feature of the Trinocchio protocol is that it enables a client, or multiple clients, to outsource a computation in a privacy-preserving way to multiple workers, while enabling fast verification of the computation. Trinocchio also ensures input independence, which means that any input party cannot let its input depend on that of another input party (e.g. by copying the input).

Informally, the Trinocchio system expands on Pinocchio as follows (note: notation is consistent with the Pinocchio protocol described in 9.2.3):

- Protocol allows multiple input parties, workers and clients (output parties);
- Protocol ensures privacy of inputs and outputs (I/O) by generalizing the single Pinocchio proof, π, to multiple proof *blocks* π_i for $i \in 1, ..., N$, where N equals the number of input and output parties (original idea of blocks explained in the next paragraph);[3]
- π_i: Includes proof terms restricted to a subset of wires, corresponding to the inputs of party i;
- Pinocchio's $KeyGen$ algorithm is adapted to $MultiKeyGen$:
  - Evaluation key EK becomes the set $\{BEK_i\}$ and verification key VK becomes set $\{BVK_i\}$ that correspond to proof block π_i, each having their own random inputs;
  - Keys $\{BEK_i\}$ and $\{BVK_i\}$ only include EK and VK terms for relevant wires, i.e. relevant to party i;
- Input phase is expanded to include public broadcasting of cryptographic commitments of inputs, to ensure input independence.

The concept of blocks from Trinocchio [SVdV16] originated from the Geppetto paper [CFH$^+$14]. Geppetto introduces a feature to enable more efficient compilation to quadratic programs, such that the prover only needs to prove correctness of used portions of the circuit. In Trinocchio, keys only contain group elements for the relevant subset of wires that correspond to the I/O parties associated with those keys.

The Trinocchio protocol consists of four polynomial-time algorithms:

- *Generate Function Evaluation and Verification Keys*
  - Trusted party creates keys for the cryptographic commitments, which are "mixed commitments"[4];
  - Trusted party creates public Evaluation and Verification Keys;
  - Trusted party throws away trapdoor information;
- *Commit to and provide input*
  - Input parties post commitment to their input blocks (needed for input independence);
  - Input parties open commitments for client(s) to verify, then provide secret-shared inputs to workers (MPC based on Shamir secret sharing, multiplication protocol from Gennaro et al. [GRR98]);
  - Workers check if shares correspond to the broadcast blocks;
- *Compute and prove*
  - Workers compute function F, produce Pinocchio proof of correct computation;
  - Calculation of polynomial h mostly local by using *Fast Fourier Transforms*;
  - Computation over bilinear group elements all performed locally;
  - Workers communicate shares of function output to the client(s);
  - Workers then post the shares of the proof elements to the bulletin board (randomized for ZK);
- *Verify*
  - Client(s) obtain their results and verify them w.r.t. information on the bulletin board.

For complete protocol details and proofs, we refer to Appendix B of [SVdV16].

[3] Please note that "blocks" in the proof of a Trinocchio protocol run are different from "blocks" in the consensus protocol of a blockchain.

[4] In such a scheme, commitment keys can be generated in two ways: such that the scheme is perfectly binding or perfectly hiding. Keys generated in both ways should be indistinguishable.

9.3.2 Geppetri: Reusable Setup for Different Computations with Adaptive zk-SNARKs

Geppetri, the successor of Trinocchio, adds valuable properties to the previous protocol. First, Geppetri enables efficient proofs that can refer to committed data. Second, Geppetri allows different verifiable computations on that data while reusing the trusted setup to generate the CRS. This latter property is also referred to as "adaptive zk-SNARK" [Lip14], i.e. the trusted CRS-generation setup is independent of the computation instance and can be reused. (Note, however, that once the function F is given, another trusted step is required to generate the function-dependent key material. While this step generates a trapdoor as well, it can only compromise the security of the specific computation instance F.) We will further explain this setup below.[5]

To solve the Billionaires' Problem, we assume the inputs to be cryptographic commitments of tax returns and require efficient verifiable computations with those commitments. Furthermore, in the blockchain context with persistent data, efficient reuse of that data and setup for different computations is highly beneficial. The trusted setup needs to be done securely, so it is preferably done only once. By using "adaptive zk-SNARKs" Geppetri achieves this. However, after generating the CRS, one more trapdoor-generating step is required to generate key material for the specific function to be computed. With knowledge of the latter trapdoor, an attacker could generate false proofs for that specific instance, but not for other computations on the committed data in general.

Geppetri requires a first setup step that produces a CRS and commitment keys. After this step, the CRS and commitment keys are reused by clients to generate evaluation and verification keys per computation. In summary, following [Lip14] and [Vee17]:

- *Generate CRS and Commitment Keys*: Outputs an extractable trapdoor commitment scheme family, consisting of:
  - a CRS and trapdoor,
  - commitment keys and trapdoor, and
  - a commitment for a given key.
- *Generate Evaluation and Verification Keys*: outputs
  - evaluation keys for the specific statement (dependent on the function instance F),
  - verification keys for the specific statement,
  - a trapdoor given the aforementioned CRS and commitment keys.

Verification in the Geppetri protocol differs slightly from the Trinocchio protocol. Recall from Section 9.2.3 that in Trinocchio, verification consists of the following steps: steps V, W, Y to check that the proof is a proof of knowledge of the witness c, a step Z to check that the same witness was used for steps V, W, Y, and finally step H to check that $p = h \cdot t$ holds. In Geppetri, where we have external commitments, commitments are checked in the V step, now renamed to "(V, C) step", and a new Z verification step ensures consistency between the commitment and the witnesses used across wires.

In the next section, we present our high-level scheme to conduct Verifiable MPC with blockchain and lay out a solution sketch to the Billionaires' Problem.

9.4 Verifiable MPC with Blockchain

We have introduced the necessary building blocks to construct a Verifiable MPC protocol that interacts with a blockchain as bulletin board. We illustrate this with the World's Billionaires' Problem. The scheme involves

[5] Recently, other improvements on the Pinocchio protocol have been proposed as well: [FFG+16] makes Pinocchio *adaptive* (as explained above) but not *zero-knowledge*, and Geppetto [CFH+14] (note: not "Geppetri") significantly reduces the computation cost for proofs by reusing intermediate computation state. However, the computations need to be known before committing, which formally is not "adaptive".

multiple input parties (tax-payers with tax return statements as inputs), multiple workers (MPC parties) and a *single, public result* (a list of the 400 wealthiest individuals). If needed, this can be extended to *multiple private results* for different *result parties* or to multiple computations on the same committed data.

After the setup phase, our scheme follows the three steps summarized below and illustrated in Figure 9.1. (For detailed Trinocchio steps, see Section 9.3.1.)

- Step 1 in which *input parties*:
  - Communicate secret shares of their inputs to workers;
  - Post cryptographic commitments of their inputs to the blockchain.
- Step 2 in which *workers*:
  - Compute secret shares of the output and the proof using MPC;
  - Recombine shares of output and proof and then post these to the blockchain.
- Step 3 in which a *smart contract*:
  - Receives and stores the outputs and the proofs (and potentially also verifies the proof, as explained in the next Section 9.4.1.)

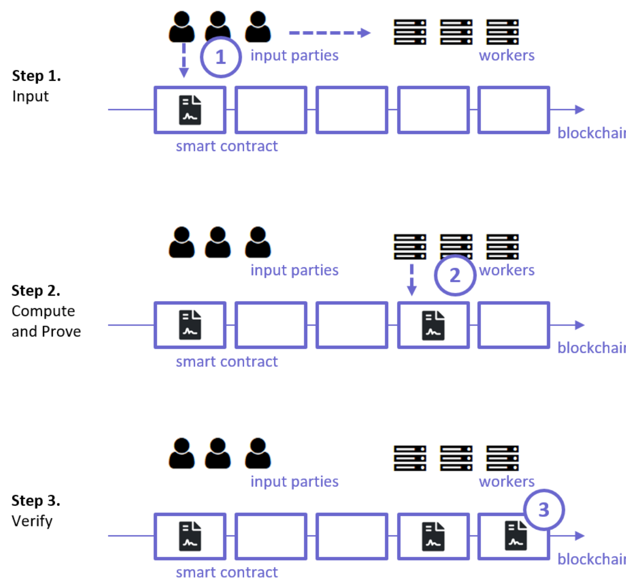


Figure 9.1: Trinocchio steps

A client that wants to verify the correctness of the output, can now download the output and proof and then run the verification step. This verification step only requires public key material.

9.4.1 Optional: Use Smart Contract to Verify Correctness Proof

To verify the proof of correct computation *on-chain* and, for example, set a flag in the public state to "valid"/"invalid", we require a blockchain that supports efficient verification of zero-knowledge proofs. This requires extending step 3 of the scheme in Figure 9.1.[6]

[6]E.g., Ethereum introduces special opcodes for zk-SNARK proof verification with Ethereum Improvement Proposal (EIP) 196 and EIP 197

Should we want to verify the correctness of the MPC computation on-chain, we could deploy a smart contract that can receive the commitments to the inputs from step 1 and the (recombined) shares of the output and the proof from step 2. It then, in step 3, verifies the correctness of the computation step, i.e., it performs the Verify algorithm from either Trinocchio or Gepepetri. After verification of the proof, the smart contract could set a flag referring to the computation to "valid", signaling to external parties that the computation was performed correctly.

9.4.2 The Billionaires' Problem: Solution Sketch

In April 2019, the world's population was estimated at 7.7 billion. Hence, to solve the Billionaires' Problem the MPC parties need to securely select the largest $k = 400$ elements out of set of size $n = 7.7 * 10^9$ and produce a proof of correct computation.

Solving the Billionaires' Problem, i.e. *privacy of inputs and computation, verifiable inputs and outputs* for $k = 400$ and $n = 7.7 * 10^9$, is not a trivial task and we invite readers to solve this challenge and share their solutions. Specifically, we suggest interested readers to review the "BOREALIS" blockchain-based scheme to compute the kth-ranked integer among sealed integers distributed among n parties [BK19]. They purposely avoid generic MPC techniques to design an efficient solution with at most four rounds of interaction. They relax a privacy constraint, as input parties learn the rank of their input. Computing pairwise comparisons in BOREALIS is based on the comparison protocol from Damgard et al. [DGK09] based on additively homomorphic encryption. Zero-knowledge proofs are based on Groth and Sahai [GS08].

As our solution is still a work-in-progress, we can only present the main elements of our scheme. First, we outline the three main (top-level) steps and then present the most important (sub-)protocols.

Scheme Steps (Top-level) To ensure *verifiable inputs* for the computation, citizens post cryptographic commitments of their tax return, signed by the tax authority, to our smart contract. As privacy should be respected for all outside the top 400, the commitments should not reveal information of individuals outside the top 400. The *verifiable output* of our protocol is a list of the top 400 wealthiest people in the world. For the individuals in our top 400, their membership to the "top 400"-set is published together with their wealth.

We recall Figure 9.1, which presents the three top-level scheme steps:

- *Input parties*, or citizens, send secret shares of their annual tax returns to the workers and post cryptographic commitments of their inputs to the blockchain.
- *Workers* compute secret shares of the output and the proof of correctness in the MPC setting. After successful computation of the secret shares of the output (our top 400 set) and the proof, *workers* recombine shares of output and proof and post it to the smart contract on the blockchain. How this top 400 set is computed securely is outlined below.
- Our *client*, which can be the general public, can now ask the smart contract to verify the proof given the commitments of the inputs, the output and the proof elements. The smart contract publicly outputs True or False, depending on whether the proof verifies.

9.4.3 Main Protocols to Compute the Top 400: Verifiable Secure Filtering and Sorting

In general, computation of the k largest items in a set of n elements corresponds to implementing a *partial sorting algorithm* or *partition-based selection*. A partial sorting algorithm is a relaxed variant of a general sorting algorithm, returning only a sorted list of the k largest items. A partition-based selection algorithm relaxes partial sorting and selects the kth largest elements, without requiring that these k elements are ordered. Linear performance can be achieved by a partition-based selection algorithm for which *quickselect* is common. Quickselect reduces average complexity to $\mathcal{O}(n)$, while worst case is $\mathcal{O}(n^2)$.

To construct a (somewhat) practical approach to compute the top 400 set, including a cryptographic proof of correctness, we strive to limit the number of communication rounds. To this end, we exploit particular properties

of our problem by first securely reducing our list of n top 400 candidates to a list of size at most $n_2 = 4000$ by applying a *verifiable secure filter*.[7] Then, we apply a *verifiable secure sorting* algorithm. Thus, assuming our workers hold secret shares of all inputs, our approach to compute the top 400 with verifiable MPC is as follows:

- Apply a *verifiable secure filter* to our full list, reducing our list from $n = 7.7 * 10^9$ to at most $n_2 = 4000$ elements.
- Apply a *verifiable secure quicksort* on the list of $n_2 = 4000$ elements.

To implement the verifiable secure n-to-n_2 filter, we proceed as follows:

- Represent the input list as a column vector L of length n, where each element L_i contains a secret shared input;
- Calculate a $n \times n_2$ "mapping" matrix M where each element $M_{i,j}$ equals $[1]$ (a sharing of 1) if item L_i is the jth occurrence of a billionaire in L and $[0]$ otherwise (it maps a billionaire in L to our smaller list);
- Securely matrix multiply $F \leftarrow L \times M$, such that the resulting length-n_2 row vector F contains the secret shared inputs for all billionaires.

The main two building blocks to construct the matrix M are a *verifiable secure "if-else"* sub-protocol, $[result] \leftarrow [c] * ([a] - [b]) + [b]$, (i.e. evaluate to $[a]$ if $[c] == [1]$, else if $[c] == [0]$ evaluate to $[b]$) and an efficient sub-protocol to create a (column) unit-vector of length n that contains a $[1]$ on the jth position if the loop passes over the jth billionaire in list L (note: our matrix M is the concatenation of these column unit-vectors).

For our verifiable secure sorting algorithm on the filtered input list F, we could pick the well-known *quicksort* algorithm by Hoare [Hoa62] as starting point. The two main sub-protocols to implement a verifiable secure version of *quicksort* are a *verifiable secure partition protocol* and a *verifiable secure comparison protocol*. The verifiable secure partition protocol is similar to our verifiable secure filter with the main difference that the input parameter is a pivot element instead of a threshold value that is independent of the elements in the set.

9.4.4 Verifiable Secure Comparison

The main bottleneck of a *verifiable secure quicksort* protocol is a *verifiable secure comparison* protocol. To reduce round complexity and computation time, we suggest the following three-step approach to compute a secret bit $[c] \leftarrow sign_bit([a] - [b])$, equivalent to the boolean $[a] < [b]$ (i.e., the sign bit equals 1 if $a < b$):

- MPC parties perform secure bit-decomposition $[c]_B = \{[c_{\ell-1}], ..., [c_0]\}$ of $[c] = [a] - [b]$, where ℓ corresponds to the bitlength of c, $c \in \{-2^{\ell-1}, ..., 2^{\ell-1} - 1\}$ and two's complement representation;
- MPC parties now select the secure most-significant bit $[c_{\ell-1}]$, which encodes whether $[a] < [b]$;
- MPC parties proof in zero-knowledge that $[c]_B$ is a bit-decomposition of $[c]$:
  - The equality $[c] = -2^{\ell-1}[c_{\ell-1}] + \sum_{i=0}^{\ell-2}[c_i]2^i$;
  - All $[c_i]$ are bits, i.e., $\forall_i [c_i](1 - [c_i]) = 0$.

Secure bit-decomposition based on the result by Schoenmakers and Tuyls [ST06] requires $\log_2 \ell + 2$ rounds. This would correspond to a $8 + 2$ round protocol in our setting with a ± 256-bit prime modulus used in the Pinocchio/Trinocchio protocols. As an alternative, one could use the constant round protocol by Reistadt and Toft [RT09] requiring 12 rounds.

To prove in zero-knowledge that $[c] = \sum_{i=0}^{\ell-1}[c_i]2^i$, the MPC parties perform verifiable secure computations and equality tests for $\sum_{i=0}^{\ell-1}[c_i]2^i = [c]$ and $\forall_i [c_i](1 - [c_i]) = 0$ using the Trinocchio (or Geppetri) protocol. As for the equality tests, we run the two-round protocol by Franklin and Haber [FH96] in the verifiable MPC setting.

[7] In 2019, Forbes reported that there are 2153 billionaires in the world by their latest count. Therefore, we are comfortable to assume that a list of 4000 elements is sufficient to capture all billionaires. See https://www.forbes.com/billionaires/.

9.4.5 The Billionaires' Problem: Next steps and Implementation Framework

The above sketch presents the most important components of a Verifiable MPC scheme to solve the Billionaires' Problem. Next steps are to extend the MPC toolkit, MPyC [Sch18], to support Verifiable MPC and implement the protocols presented in this chapter.

MPyC is a Python framework for *Shamir secret-sharing-based MPC* in the honest-majority setting. MPyC supports computations on secret-shared values and handles the message exchange between parties asynchronously. MPyC provides secure number types and secure operations that are made available through Python's mechanism for operator overloading. MPyC can be viewed as a successor to the VIFF (Virtual Ideal Functionality Framework) MPC library [Gei10].[8]

9.5 Conclusion

This chapter explained the value of Verifiable MPC in the blockchain context. We showed how Verifiable MPC with blockchain can enable use cases that require privacy of inputs and verifiability of inputs and outputs. We outlined a scheme to solve the World's Billionaires' Problem, a generalization of Yao's Millionaires' Problem that makes the entire computation verifiable. Our scheme can be extended to general *privacy-preserving and publicly verifiable outsourcing* and demonstrates the early stages of a prototype. This prototype is a stepping stone to support PRIViLEDGE use-case "UC2: Medical Insurance" and we hope it is beneficial to other (non-PRIViLEDGE) use cases as well.

[8]MPyC currently provides the following functionalities:

- Information-theoretic and pseudorandom secret sharing;
- Finite-field arithmetic;
- Secure types for secret-sharing, specifically: $SecField$, $SecInt$, $SecFxp$;
- Secure random module with functions based on Python standard library counterpart with secure type as input, and helpers: random_unit_vector, random_permutation and random_derangement;
- Secure statistics module with functions based on Python standard library counterpart including mean, median, mode, variance, and stdev.

Chapter 10

Hash Based Server Assisted Signatures

10.1 Background

All digital signature schemes in wide use today (RSA, DSA, ECDSA) are known to be vulnerable to quantum attacks by Shor's algorithm. While the best current experimental results are still toy-sized, it takes a long time for new cryptographic schemes to be accepted and deployed, so it is of considerable interest to look for post-quantum secure alternatives already now. Error-correcting codes, discrete lattices, and multi-variate polynomials have been used as foundations for proposed replacement schemes. However, these are relatively complex structures and new constructions in cryptography, so require significant additional scrutiny before gaining trust.

Hash functions, on the other hand, have been studied for decades and are widely believed to be resilient to quantum attacks. The best known quantum results against hash functions are using Grover's algorithm [Gro96] to find a pre-image of a given k-bit value in $2^{k/2}$ queries instead of the 2^k queries needed by a classical attacker, and Brassard *et al*'s modification [BHT98] to find a collision in $2^{k/3}$ instead of $2^{k/2}$ queries. To counter these attacks, it would be sufficient to deploy hash functions with correspondingly longer outputs when moving from pre-quantum to post-quantum setting. Hash function based signature schemes have indeed been studied extensively, starting with Lamport's early proposal in [DH76] and up to the current state of the art in XMSS [BDH11,HRB13, HRS16] and SPHINCS [BHH+15].

In [BLT17], we combined hash function based message authentication codes and hash-then-publish time-stamping to obtain a server assisted signature scheme. An authentication code enables the recipient of a message to verify that it was indeed sent by the claimed sender (or, more accurately, by someone who knows the sender's key) and has not been modified in transit. However, message authentication codes lack the non-repudiation property: the recipient can't prove the authenticity of the message to a third party. This is due to the symmetric nature of the codes: the recipient shares the secret key with the sender and thus could have fabricated both the message and the accompanying code.

The main idea of the BLT scheme is to use time-stamping to break the symmetry. The signer pre-generates a sequence of one-time keys z_i as unpredictable (random or pseudo-random) values, binds each key to its intended usage time t_i by computing a hash commitment $x_i = h(t_i, z_i)$, aggregates the commitments into a hash tree, and publishes the root hash of the tree as the public key (but keeps the signing keys z_i secret). To sign message M at time $T = t_i$, the signer authenticates the message with the corresponding key z_i by time-stamping the pair (M, z_i). The signature is the tuple (t_i, z_i, c_i, a_T), where c_i is the hash chain proof linking the pair (t_i, z_i) to the signer's public key and a_T is the time-stamp proving the message-key pair (M, z_i) was authenticated at time $T = t_i$. Note that it is safe for the signer to release the key z_i as part of the signature, as its intended usage time has passed and thus it can no longer be used to create more valid signatures. It is the time-stamping step that creates the crucial asymmetry: before t_i, only the signer knew the key z_i; after t_i the key is public, but any new pairs (M', z_i) could only be stamped with times $T' > t_i$.

10.2 Blockchain Backed Key State Management

The signing keys in the original BLT scheme are really not one-time, but rather time-bound: every key can be used for signing only at a specific point of time. This incurs quite a large overhead as keys must be pre-generated even for time periods when no signatures are created. In particular, key generation on smart-cards would be prohibitively slow in real-world parameters.

To avoid the inherent inefficiency of pre-generating keys for every possible time slot, in [BLT18] we proposed a model where each key could be just just once, but at the time of the signer's choosing.

A naive approach would be to have the signer time-stamp each signature, just as in the basic BLT scheme: in case of a dispute, the signature with the earlier time-stamp wins and the later one is considered a forgery. This obviously makes verification very difficult, but more importantly gives the signer a way to deny any signature: before signing a document d with a key z, the signer can use the same key to privately sign some dummy value x; when later demanded to honor the signature on document d, the signer can show the signature on x and declare the signature on d a forgery.

To prevent this, we assign every signer to a designated server which allows each key to be used only once. The easiest way to track the one-time keys is to require them to be used in order and keep an index of the last used (or next available) key for each signer. To reduce the need to trust the server (which, being in service of the signer, might be persuaded to collude), we also propose a hash tree based authenticated data structure that allows a consortium of notaries to keep just one hash commitment per monitored server and use that to verify the validity of each key state transition. The update of the commitment is allowed only if sufficient majority of the notaries approves it as valid.

Furthermore, the updates can be batched in such a way that only the final state of the commitment after a batch of updates needs to be published and every signature issued within the batch can be linked to the final commitment. Given all of the above, a distributed ledger is perfect for publishing these batch commitments to use them as trust anchors for signature verification. The batch commitments, aggregating key transitions across all the signers supported by a server, do not leak any information about the activity of any individual signer, except possibly the fact that nothing was signed in case the commitment does not change over some period of time (the latter can easily be mitigated by having the server update a dummy counter in each round when there are no actual signing requests to be processed).

10.3 Forward-Resistant Tag Systems

In [BFL+19a] and [BFL+19b], we used the observation that the time-stamp component of a BLT signature, being back-dating resistant, already prevents re-binding of the signing key to an earlier time, so it is sufficient for the remaining signature components to only prevent re-binding to a later time. This gave rise to the new notion of *forward-resistant tags*. We gave formal definition of such tags, which could in some ways be seen as half-signatures, and proved that combining a forward-resistant tag system with back-dating resistant time-stamping indeed yields a provably secure signature scheme.

Binding each key to a fixed time, like it was done in the original BLT scheme from [BLT17], satisfies the requirements of a forward-resistant tag system, but is overly restrictive. In [BFL+19a] and [BFL+19b], we take advantage of the relaxed requirements to define several other, more flexible tag systems and show that they induce signature schemes more efficent than the one from [BLT17] and with lower trust requirements on the server than the one from [BLT18].

A limitation of the results of [BFL+19a] and [BFL+19b] is an overly simplistic model of the time-stamping service, which is assumed to be a plain-text log of all time-stamped datums. In all practical systems, only hash values are sent to the time-stamping service, and in most hash-then-publish systems (see [BKL13], for example), the client requests are further aggregated before the publishing step takes place.

In [BFLTa], we undertake a systematic study of different models of hash-then-publish time-stamping, to establish the exact requirements placed by each of them on the properties of the tag system on one hand, and on

the hash functions on the other hand, for the resulting signature scheme to remain provably secure. In [BFLTb] we report on the formalization and machine-checking of several of these results using the EasyCrypt framework [BDG+13].

Chapter 11

Conclusion

This document presented recent advances by PRIViLEDGE partners in core protocol design for distributed legders. All Chapters focused on contemporary DLT challenges: privacy, trust minimization, scalability and security - to name the most important ones.

Common research themes in this document are public verifiability, efficiency and security properties of zero knowledge protocols, consensus protocol design and formal security proofs of these protocols (typically in the UC model). We believe this aligns well with the research focus of the wider privacy-enhancing cryptography and DLT communities, and is therefore a valuable contribution to these communities.

The next steps for WP3 are to revise and finalize the core protocols and test them against the selected use cases. The next and final public deliverable for WP3 is Deliverable D3.3 “Revision of Extended Core Protocols” by month 36 of the project, i.e., end-of-year 2020.

Bibliography

[AB19] Shahla Atapoor and Karim Baghery. Simulation extractability in Groth's zk-SNARK. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26-27, 2019, Proceedings*, pages 336–354, 2019.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proc. 13th European Conference on Computer Systems (EuroSys)*, pages 30:1–30:15, April 2018.

[ABC+15] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2015:1192, 2015.

[ABL+19a] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, Janno Siim, and Michal Zajac. DL-extractable UC-commitment schemes. In *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, pages 385–405, 2019.

[ABL+19b] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, Janno Siim, and Michal Zajac. UC-secure CRS generation for snarks. In *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, pages 99–117, 2019.

[ABLZ17] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, and Michal Zajac. A subversion-resistant SNARK. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part III*, pages 3–33, 2017.

[ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE, 2014.

[AKM+15] Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis, editors, *Proc. Trust and Trustworthy Computing (TRUST)*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.

[Bag19a] Karim Baghery. On the efficiency of privacy-preserving smart contract systems. In *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings*, pages 118–136, 2019.

[Bag19b] Karim Baghery. Subversion-resistant simulation (knowledge) sound NIZKs. Cryptology ePrint Archive, Report 2019/1162, 2019. http://eprint.iacr.org/2019/1162.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 566–582. Springer, 2001.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 326–349. ACM, 2012.

[BCD+14] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. 2014. http://www.opensciencereview.com/papers/123/enablingblockchain-innovations-with-pegged-sidechains.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459–474. IEEE Computer Society, 2014.

[BCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 287–304. IEEE Computer Society, 2015.

[BCNP04] Boaz Barak, Ran Canetti, Jesper Buus Nielsen, and Rafael Pass. Universally composable protocols with relaxed set-up assumptions. In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 186–195. IEEE Computer Society, 2004.

[BCTV13] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive arguments for a von neumann architecture. Cryptology ePrint Archive, Report 2013/879, 2013. http://eprint.iacr.org/2013/879.

[BDG+13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, volume 8604 of *LNCS*, pages 146–166. Springer, 2013.

[BDH11] Johannes A. Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS—A practical forward secure signature scheme based on minimal security assumptions. In *PQCrypto 2011, Proceedings*, volume 7071 of *LNCS*, pages 117–129. Springer, 2011.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. Cryptology ePrint Archive, Report 2014/075, 2014. https://eprint.iacr.org/2014/075.

[BFL+19a] Ahto Buldas, Denis Firsov, Risto Laanoja, Henri Lakk, and Ahto Truu. A new approach to constructing digital signature schemes (extended paper). Cryptology ePrint Archive, Report 2019/673, 2019. https://eprint.iacr.org/2019/673.

[BFL+19b] Ahto Buldas, Denis Firsov, Risto Laanoja, Henri Lakk, and Ahto Truu. A new approach to constructing digital signature schemes (short paper). In *IWSEC 2019, Proceedings*, volume 11689 of *LNCS*, pages 363–373. Springer, 2019.

[BFLTa] Ahto Buldas, Denis Firsov, Risto Laanoja, and Ahto Truu. On tag systems and time-stamping. Unpublished manuscript.

[BFLTb] Ahto Buldas, Denis Firsov, Risto Laanoja, and Ahto Truu. Verified security of BLT signature scheme. To appear at ACM SIGPLAN CPP 2020.

[BFS16] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. Nizks with an untrusted CRS: security in the face of parameter subversion. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 777–804, 2016.

[BGG17] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. *IACR Cryptology ePrint Archive*, 2017:602, 2017.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 913–930, 2018.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 1–10, New York, 1988. ACM.

[BHH+15] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. In *EUROCRYPT 2015, Proceedings, Part I*, volume 9056 of *LNCS*, pages 368–397. Springer, 2015.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013.

[BHT98] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN'98, Proceedings*, volume 1380 of *LNCS*, pages 163–169. Springer, 1998.

[BK19] Erik-Oliver Blass and Florian Kerschbaum. Borealis: Building block for sealed bid auctions on blockchains. Cryptology ePrint Archive, Report 2019/276, 2019. https://eprint.iacr.org/2019/276.

[BKL13] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. Keyless signatures' infrastructure: How to build global distributed hash-trees. In *NordSec 2013, Proceedings*, volume 8208 of *LNCS*, pages 313–320. Springer, 2013.

[BLT17] Ahto Buldas, Risto Laanoja, and Ahto Truu. A server-assisted hash-based signature scheme. In *NordSec 2017, Proceedings*, volume 10674 of *LNCS*, pages 3–17. Springer, 2017.

[BLT18] Ahto Buldas, Risto Laanoja, and Ahto Truu. A blockchain-assisted hash-based signature scheme. In *NordSec 2018, Proceedings*, volume 11252 of *LNCS*, pages 138–153. Springer, 2018.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356. Springer, 2017.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In Yvo Desmedt, editor, *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2003.

[BPS16] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

[Bra87] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.

[BSBHR17] Eli Ben-Sasson, Iddo Bentov, Ynon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Manuscript.(2017). Slides at https://people. eecs. berkeley. edu/˜ alexch/docs/pcpip_bensasson. pdf*, 2017.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. *IACR Cryptology ePrint Archive*, 2014:349, 2014.

[BSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM J. Comput.*, 20(6):1084–1118, 1991.

[Buc16] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[But14] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[BW06] Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 290–307. Springer, 2006.

[Can01] Ran 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*, pages 136–145. IEEE Computer Society, 2001.

[CBPS10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 11–19, New York, 1988. ACM.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[CFH+14] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. Cryptology ePrint Archive, Report 2014/976, 2014. https://eprint.iacr.org/2014/976.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In Jeffrey Scott Vitter, editor, *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 209–218. ACM, 1998.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[CHK03] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 255–271. Springer, 2003.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 78–96. Springer, 2006.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503. ACM, 2002.

[CM18] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. e-print, arXiv:1802.07242 [cs.DC], 2018.

[Com18] Cardano Community. Cardano settlement layer documentation. https://cardanodocs.com/technical/, October 18 2018.

[CPS07] Ran Canetti, Rafael Pass, and Abhi Shelat. Cryptography from sunspots: How to use an imperfect reference string. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 249–259. IEEE Computer Society, 2007.

[CS14a] Chris Culnane and Steve Schneider. A peered bulletin board for robust use in verifiable voting systems. *CoRR*, abs/1401.4151, 2014.

[CS14b] Chris Culnane and Steve A. Schneider. A peered bulletin board for robust use in verifiable voting systems. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 169–183, 2014.

[CV17] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. In Andréa W. Richa, editor, *Proc. 31st Intl. Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, 2017.

[Dam] Ivan Damgård. Towards Practical Public Key Systems Secure against Chosen Ciphertext Attacks. pages 445–456.

[DDFN07] Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In Kaoru Kurosawa, editor, *Advances in Cryptology: ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.

[DFGK14] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2014.

[DG03] Ivan Damgård and Jens Groth. Non-interactive and reusable non-malleable commitment schemes. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 426–437. ACM, 2003.

[DGHM13] Grégory Demay, Peter Gazi, Martin Hirt, and Ueli Maurer. Resource-restricted indifferentiability. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 664–683. Springer, 2013.

[DGK09] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. A correction to 'efficient and secure comparison for on-line auctions'. *IJACT*, 1(4):323–324, 2009.

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98. Springer, 2018.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[DN92] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.

[DPSZ11] I. Damgård, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

[DPW+16] Johnny Dilley, Andrew Poelstra, Jonathan Wilkins, Marta Piekarska, Ben Gorlick, and Mark Friedenbach. Strong federations: An interoperable blockchain solution to centralized third party risks. *CoRR*, abs/1612.05491, 2016.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, volume 3386 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 2005.

[FFG+16] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. Cryptology ePrint Archive, Report 2016/985, 2016. https://eprint.iacr.org/2016/985.

[FGP14] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. Cryptology ePrint Archive, Report 2014/202, 2014. https://eprint.iacr.org/2014/202.

[FH96] Matthew K. Franklin and Stuart Haber. Joint encryption and message-efficient secure computation. *J. Cryptology*, 9(4):217–232, 1996.

[Fuc18] Georg Fuchsbauer. Subversion-zero-knowledge snarks. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 315–347. Springer, 2018.

[Gei10] Martin Geisler. *Cryptographic Protocols: Theory and Implementation*. PhD thesis, Aarhus University, 2010.

[GG17] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, pages 529–561, 2017.

[GG18] Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine quorum systems. In *Proc. 22nd Conference on Principles of Distributed Systems (OPODIS)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, 2018.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 465–482, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the IACR Eurocrypt Conference*. International Association for Cryptologic Research, May 2013.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating System Principles (SOSP)*, pages 150–162, 1979.

[GK03] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the fiat-shamir paradigm. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 102–113. IEEE Computer Society, 2003.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 291–323, 2017.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, pages 698–728, 2018.

[GKP+13] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 555–564, New York, NY, USA, 2013. ACM.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 581–612. Springer, 2017.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game - or - a completeness theorem for protocols with honest majority. In *Proc. 19th Symposium on Theory of Computing (STOC '87)*, pages 218–229, New York, 1987. ACM.

[GO07] Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007, 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007, Proceedings*, volume 4622 of *Lecture Notes in Computer Science*, pages 323–341. Springer, 2007.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for NP. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 339–358. Springer, 2006.

[GOT18] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. Cryptology ePrint Archive, Report 2018/1105, 2018. https://eprint.iacr.org/2018/1105.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM STOC, Proceedings*, pages 212–219. ACM, 1996.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In Xuejia Lai and Kefei Chen, editors, *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *Lecture Notes in Computer Science*, pages 444–459. Springer, 2006.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 321–340, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Report 2016/260, 2016. https://eprint.iacr.org/2016/260.

[GRR98] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *17th annual ACM symposium on Principles of Distributed Computing (PODC '98)*, pages 101–111, New York, 1998. ACM.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2008.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 99–108. ACM, 2011.

[HBHW18] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. 2018.

[HM00] Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[Hoa62] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[HRB13] Andreas Hülsing, Lea Rausch, and Johannes A. Buchmann. Optimal parameters for XMSS MT. In *CD-ARES 2013, Proceedings*, volume 8128 of *LNCS*, pages 194–208. Springer, 2013.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Fang Song. Mitigating multi-target attacks in hash-based signatures. In *PKC 2016, Proceedings, Part I*, volume 9614 of *LNCS*, pages 387–416. Springer, 2016.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

[JKS16] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 283–295, 2016.

[JM03] Flavio P. Junqueira and Keith Marzullo. Synchronous consensus for dependent process failures. In *Proc. 23rd Intl. Conference on Distributed Computing Systems (ICDCS)*, 2003.

[JMHD10] Flavio P. Junqueira, Keith Marzullo, Maurice Herlihy, and Lucia Draque Penso. Threshold protocols in survivor set systems. *Distributed Computing*, 23:135–149, 2010.

[KC18] Aggelos Kiayias and Michele Ciampi. State of the art of cryptographic ledgers. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, 2018. http://priviledge-project.eu/publications/deliverables.

[KDF13] Joshua A. Kroll, Ian C. Davey, and Edward W. Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *The Twelfth Workshop on the Economics of Information Security (WEIS 2013), Washington DC*, June 10-11 2013.

[KFTS17] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of monero's blockchain. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, pages 153–173, 2017.

[KJG+18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598. IEEE Computer Society, 2018.

[KKL+18] Aggelos Kiayias, Annabell Kuldmaa, Helger Lipmaa, Janno Siim, and Thomas Zacharias. On the security properties of e-voting bulletin boards. In *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, pages 505–523, 2018.

[KL11] Dafna Kidron and Yehuda Lindell. Impossibility results for universal composability in public-key models and with fixed inputs. *J. Cryptology*, 24(3):517–544, 2011.

[KLS16] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. Proofs of proofs of work with sublinear complexity. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2016.

[KMS+15] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Report 2015/675, 2015. https://eprint.iacr.org/2015/675.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498. Springer, 2013.

[KMZ17] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. *IACR Cryptology ePrint Archive*, 2017:963, 2017.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

[Kri18] Toomas Krips. State of the art on privacy-enhancing cryptography for ledgers. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, 2018. http://priviledge-project.eu/publications/deliverables.

[KYMM18] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in zcash. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 463–477, 2018.

[KZM+15a] Ahmed E. Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T.-H. Hubert Chan, Charalampos Papamanthou, Rafael Pass, Abhi Shelat, and Elaine Shi. C∅C∅: A Framework for Building Composable Zero-Knowledge Proofs. Technical Report 2015/1093, November 10, 2015. http://eprint.iacr.org/2015/1093, last accessed version from 9 Apr 2017.

[KZM+15b] Ahmed E. Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T.-H. Hubert Chan, Charalampos Papamanthou, Rafael Pass, Abhi Shelat, and Elaine Shi. How to use snarks in universally composable protocols. *IACR Cryptology ePrint Archive*, 2015:1093, 2015.

[Lam86] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 86–101, 1986.

[Ler16] Sergio Demian Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016.

[LHL94] Chuan-Ming Li, Tzonelih Hwang, and Narn-Yih Lee. Threshold-multisignature schemes where suspected forgery implies traceability of adversarial shareholders. In Alfredo De Santis, editor, *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 1994.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 169–189. Springer, 2012.

[Lip14] Helger Lipmaa. Prover-efficient commit-and-prove zero-knowledge snarks. Cryptology ePrint Archive, Report 2014/396, 2014. https://eprint.iacr.org/2014/396.

[Lip19] Helger Lipmaa. Simulation-extractable SNARKs revisited. Cryptology eprint archive, report 2019/612, July 2019.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 17–30, 2016.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[Max13] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. https://bitcointalk.org/?topic=279249, August 2013.

[Maz16] David Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Stellar, available online, https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. *IACR Cryptology ePrint Archive*, 2019:99, 2019.

[Mic16] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[MPJ+16] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. *Commun. ACM*, 59(4):86–93, 2016.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *IEEE FOCS*, pages 120–130, 1999.

[MSH$^+$18] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. An empirical analysis of traceability in the monero blockchain. *PoPETs*, 2018(3):143–163, 2018.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[Nao03] Moni Naor. On cryptographic assumptions and challenges. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 2003.

[Noe15] Shen Noether. Ring signature confidential transactions for monero. *IACR Cryptology ePrint Archive*, 2015:1098, 2015.

[NVV18] Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 65–80. USENIX Association, 2018.

[NW98] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

[PBF$^+$18] Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. Confidential assets. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, volume 10958 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2018.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 238–252. IEEE Computer Society, 2013.

[PS17] Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, pages 380–409, 2017.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, 2017.

[Rip19] Ripple. XRP ledger documentation ≫ Concepts ≫ Technical FAQ. Available online, https://developers.ripple.com/technical-faq.html, 2019.

[RMK14] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 345–364, 2014.

[RS13] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 6–24, 2013.

[RT09] Tord Ingolf Reistad and Tomas Toft. Linear, constant-rounds bit-decomposition. In Dong Hoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 245–257. Springer, 2009.

[RY07] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. Cryptology ePrint Archive, Report 2007/264, 2007. https://eprint.iacr.org/2007/264.

[Sch18] Berry Schoenmakers. MPyC secure multiparty computation in python. https://github.com/lschoe/mpyc, 2018.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[ST87] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[ST06] B. Schoenmakers and P. Tuyls. Efficient binary conversion for Paillier encryptions. In *Advances in Cryptology—EUROCRYPT '06*, volume 4004 of *LNCS*, pages 522–537, Berlin, 2006. Springer.

[Ste15] Stellar. On worldwide consensus. Available online, https://medium.com/a-stellar-journey/on-worldwide-consensus-359e9eb3e949, 2015.

[SV15] Berry Schoenmakers and Meilof Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security*, pages 3–22, Cham, 2015. Springer International Publishing.

[SV18] Luisa Siniscalchi and Ivan Visconti. Definitions and notions of privacy-enhancing cryptographic primitives for ledgers. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, 2018. http://priviledge-project.eu/publications/deliverables.

[SVdV16] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, volume 9696 of *Lecture Notes in Computer Science*, pages 346–366. Springer, 2016.

[SYB14] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs, available online, https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[Szt15] Paul Sztorc. Drivechain - the simple two way peg, November 2015. http://www.truthcoin.info/blog/drivechain/.

[TS] Stefan Thomas and Evan Schwartz. A protocol for interledger payments. https://interledger.org/interledger.pdf.

[Vee17] Meilof Veeningen. Pinocchio-based adaptive zk-snarks and secure/correct adaptive function evaluation. In Marc Joye and Abderrahmane Nitaj, editors, *Progress in Cryptology - AFRICACRYPT 2017 - 9th International Conference on Cryptology in Africa, Dakar, Senegal, May 24-26, 2017, Proceedings*, volume 10239 of *Lecture Notes in Computer Science*, pages 21–39, 2017.

[vS13] Nicolas van Saberhagen. Cryptonote v 2.0. https://cryptonote.org/whitepaper.pdf, October 17 2013.

[WLB14] Shawn Wilkinson, Jim Lowry, and Tome Boshevski. Metadisk a blockchain-based decentralized file storage application. *Storj Labs Inc., Technical Report, hal*, pages 1–11, 2014.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[Woo16] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework, 2016.

[Woo19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Technical Report aeeda84, July 2019. Byzantium version.

[Yao82] A. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] A. Yao. How to generate and exchange secrets. In *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS '86)*, pages 162–167. IEEE Computer Society, 1986.

[Zam17] Vlad Zamfir. Casper the friendly ghost: A "correct-by-construction" blockchain consensus protocol. https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf, December 17 2017.

[ZMR18] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 931–948. ACM, 2018.

[ZSJ+18] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar R. Weippl, and William J. Knottenbelt. A wild velvet fork appears! inclusive blockchain protocol changes in practice - (short paper). In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, volume 10958 of *Lecture Notes in Computer Science*, pages 31–42. Springer, 2018.