DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D2.3 – Improved Constructions of Privacy-Enhancing Cryptographic Primitives for Ledgers

Due date of deliverable: 31 December 2020
Actual submission date: 31 December 2020

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

Lead contractor: Guardtime AS
Duration: 42 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	

D2.3

Improved Constructions of Privacy-Enhancing Cryptographic Primitives for Ledgers

Editor
Vincenzo Iovino, Ivan Visconti (UNISA)

Contributors
Angelo De Caro (IBM)
Michele Ciampi (UEDIN)
Berry Schoenmakers, Toon Segers (TUE)
Mikhail Volkhov, Markulf Kohlweiss (UEDIN)
Ahto Truu, Henri Lakk (GT)

Reviewers
Sven Heiberg (SCCEIV)
Michele Ciampi, Markful Kohlweiss (UEDIN)

31 December 2020
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.

Executive Summary

This document focuses on improved constructions of cryptographic primitives that are relevant to distributed ledger technologies (DLTs). The document consists of several technical contributions of partners of PRIViLEDGE that in the first 3 years of the project have conducted research stimulated also by inputs received from other work packages and from deliverable D2.2. In particular, the document introduces a transform to preserve security of smart contracts in the presence of forks in blockchains, privacy-preserving auditable token payment systems designed for permissioned blockchains, a new zero-knowledge proof systems using blockchains as setup, timed cryptographic primitives, secure groups, and a family of signature schemes. All such constructions are relevant building blocks for distributed ledger technologies and their applications.

Contents

Chapter 1

Introduction

The rise of distributed ledger technologies (DLTs) introduces new challenges for preserving data privacy and integrity and motivates the need of new security notions and corresponding cryptographic building blocks. Indeed, classical privacy-preserving cryptographic primitives do not exploit distributed ledgers, often rely on centralized components and in some cases are not optimized for DLT applications. This document, through several chapters briefly described below, presents new cryptographic primitives that are useful for DLTs and their applications.

Publicly verifiable zero knowledge. Chapter 2 provides a publicly verifiable zero-knowledge proof system based on any blockchain that, very roughly, satisfies the following unpredictability property. Sufficiently many future honest blocks added to the blockchain contain a high min-entropy string in a specific location. This new proof system is secure against a verifier/prover that can corrupt blockchain players adaptively. In particular, it remains zero knowledge even if the blockchain eventually collapses and all blockchain players are controlled by the zero-knowledge adversary.

Quick execution on blockchain. Chapter 3 shows a general transform to design smart contracts that retain security in the presence of forks. The security notion used for modeling executions of smart contracts is secure multi-party computation (MPC). The transform satisfies fairness with penalties, therefore a player that aborts the execution of a smart contract can be financially penalized. A consequence of this result is that blockchain players do not need to wait for many confirmations and therefore using this transform one can design protocols that are significantly faster than previous constructions.

Privacy-preserving auditable token payments in a permissioned blockchain system. Token management systems (e.g., distributed payment systems) were the first application of blockchain technology and are typically implemented in permissionless blockchains like Monero and Zerocash and without a satisfying level of privacy in Bitcoin. Chapter 4 presents a privacy-preserving token management system that is designed for permissioned blockchain systems and supports fine-grained auditing. The scheme is secure under computational assumptions in bilinear groups, in the random-oracle model.

Timed primitives and secure timestamping. Chapter 5 shows how to realize in the Universal Composability (UC) model the zero-knowledge and signature functionalities proposed in Chapter 5 of deliverable D2.2. The first part of the chapter provides a high-level overview of the functionalities and of their UC-realization. The remaining part of the chapter formally shows how to realize these functionalities.

Succinct Non-Interactive Arguments of Knowledge (SNARKs) from linear-Size universal and updateable structured reference strings. Chapter 6 presents Sonic, a new zk-SNARK for general arithmetic circuit satisfiability. Sonic requires a trusted setup, but unlike conventional SNARKs the structured reference string supports all circuits (up to a given size bound) and is also updatable, so that it can be continually strengthened. This addresses many of the practical challenges and risks surrounding such setups. The structured reference string in Sonic also does not need to be specialized or pre-processed for a given circuit. This makes a large, distributed and never-ending setup process a practical reality.

Secure groups. Chapter 7 introduces a scheme to implement finite groups as oblivious data structures, meaning that no information can be inferred about the values of the group elements after a sequence of operations.

For a given group, the scheme defines the oblivious representation of group elements and oblivious operations on group elements. Operations include the group law, exponentiation and inversion, random sampling and encoding/decoding. The goal of the chapter is to show that secure groups are a convenient and powerful abstraction to develop finite group-based applications on top of MPC, in particular cryptographic applications. To illustrate how to construct cryptosystems with secure groups, the chapter includes an implementation of a threshold cryptosystem using groups of prime order with the MPyC framework [Sch18] that is relevant for private computations with impacts on smart contracts. Two practical applications of this threshold cryptosystem are a protocol to convert ciphertexts to Shamir shares, and a protocol to convert Shamir shares to ciphertexts.

Server-Assisted Hash-Based Signatures. Chapter 8 introduces a new family of hash-based signature schemes. A novel design element of such schemes is the reliance on a time-stamping service as an inherent component. The performance of the new schemes is very competitive, but the reliance on a time-stamping service adds dependence on a security-critical external component. To reduce the need to trust the time-stamping service provider, the chapter suggests the use of hash-then-publish time-stamping schemes, which in many ways could be considered ancestors of modern blockchain systems.

Chapter 2

Publicly Verifiable Zero Knowledge from Blockchains

In TCC 2017 Goyal and Goyal proposed the first – and currently only– construction of a publicly verifiable zero-knowledge (pvZK) proof system that leverages a blockchain as setup assumption. Such construction can be instantiated only through proof-of-stake blockchains and presents a few more limitations and assumptions: (1) the adversary can only perform static corruption of the stakeholders, (2) keys of the stakeholders must also allow for encryption and (3) honest stakeholders must never leak their secret keys (even when no stake is left with respect to those keys).

In this chapter we provide a publicly verifiable zero-knowledge proof system, based on *any* blockchain (not only proof-of-stake) that, very roughly, satisfies the following unpredictability property. Sufficiently many future honest blocks added to the blockchain contain a high min-entropy string in a specific location (e.g., a new wallet for cashing the mining reward). Our proof system is secure against a verifier/prover that can corrupt blockchain players adaptively. In particular, it remains zero knowledge even if the blockchain eventually collapses and all blockchain players are controlled by the zero-knowledge adversary.

The full version of the results presented in this chapter can be found here [SSV20].

2.1 Introduction

Following the success of Bitcoin many other cryptocurrencies based on blockchain technology have been proposed and, despite a few security issues, they are still expanding their networks with gigantic market capitalizations. What is so appealing in decentralized blockchains?

Public verifiability. One of the most supported answers is the paradigm shift from trust in some entity to "public verifiability". This property allows every one to check that the system works consistently with the prespecified rules of the game. This makes users willing to be involved in transactions recorded in a blockchain therefore investing their real-world money. In many blockchain applications both anonymity and public verifiability are required, calling for advanced cryptographic primitives such as publicly verifiable zero-knowledge proofs. For example, when the blockchain is used to record payments, confidential transactions are indeed implemented using publicly verifiable zero-knowledge proofs called zk-SNARKs [BCGV16, GGPR13] (e.g., ZCash [ZCa]).

Publicly verifiable zero-knowledge proofs. Known constructions of publicly verifiable zero-knowledge (pvZK) proofs are instantiated with *non-interactive* zero-knowledge proofs (NIZK) and, as such, require setup assumptions. Indeed, despite a significant effort of the research community, constructions of NIZK proofs either rely on the existence of a trusted common reference string (CRS) computed by a trusted entity or are based on heuristic

assumptions (e.g., random oracles). Recent exiting work has shown mechanism to relax the trust assumptions required to generate the CRS [BGG18] or to mitigate the effect of a malicious CRS [GKM+18a, MBKM19]. While this line of work is very promising, it still requires the employment of third entities that help computing the CRS.

Publicly verifiable zero-knowledge proofs from a "Blockchain Assumption". Since its introduction in 2008 with Nakamoto's protocol [Nak08a], blockchain protocols have been scrutinized by many communities, and currently, we have a good understanding of the security properties they provide and the class of adversaries they withstand. In particular, several work from the cryptographic community provided a formalization of the Bitcoin security guarantees [GKL15, PSS17], a formalization of the ideal functionality it implements [BMTZ17] as well as game-theoretic analysis [BGM+18]. Furthermore, new blockchain designs have been proposed, based on different assumptions on the collective power of the adversary. Some prominent examples that are also implemented in practice are Ouroboros [BGK+18] and Algorand [GHM+17].

Given that blockchains have been formally analyzed and are up and running in practice, a natural question to ask is whether we can use blockchain as a setup assumption *to replace* trusted setups required for certain cryptographic tasks, particularly, for publicly verifiable zero-knowledge proof systems that are needed the most in blockchain applications.

This question was first investigated by Goyal and Goyal in [GG17], where they aimed to construct NIZK using as setup the existence of a proof-of-stake (PoS) blockchain. The security of the NIZK proof provided in [GG17] – that we will denote by GG-NIZK– however is analyzed in a threat model that does not faithfully match the threat model of PoS blockchains, since it considers only static adversaries, and additionally it poses restrictions on honest stakeholder, that are not necessarily contemplated by a general PoS blockchain. Specifically, the zero-knowledge property of GG-NIZK is proved in presence of a *static adversary* who decides in advance which stakeholder will corrupt in its entire attack. This does not match the threat model for proof-of-stake blockchain where an adversary is allowed to corrupt stakeholders at any time, and the only restriction is that, at any point, the total amount of stake held by the adversary is a minority of the total stake of the system. Moreover, in the GG-NIZK security analysis, the zero-knowledge property holds under the additional assumption that honest stakeholders will never leak their stakeholder keys, not even when such keys become irrelevant for the blockchain protocol (for example, because there is zero stake associated to it).

2.1.1 Publicly Verifiable Zero Knowledge from Blockchains

A recent work by Choudhuri et al. [CGJ19] shows that using a blockchain as a black-box object that provides only a global ledger does not allow to overcome some impossibility results in the plain model and in particular it does not allow to construct pvZK proofs. Therefore, in order to build a publicly verifiable zero-knowledge proof system from a blockchain, it seems that one needs to leverage some "structure" of the blockchain and/or provide more power to the simulator besides black-box access to a global ledger. Following [GG17] we will assume that the simulator has the power of controlling the honest parties. However, unlike [GG17] we assume that the adversary can adaptively corrupt players and moreover we want our pvZK proof to remain zero knowledge even in case of blockchain failure, in the sense that in the future the adversary might take full control over the blockchain.

To leverage this power, we assume a blockchain that has a mild structure. Very informally (a formal definition is provided in Assumption 1) we assume the following. First, every block contains a distinguished field v. For concreteness, the reader can assume that this field is the same as the "coinbase" value of any Bitcoin block, and to ease the discussion, in the text that follows, we will call this field wallet. Our blockchain assumption, very roughly, is that there exist a parameter t, such that, for any sequence of t blocks, considering the new wallets[1] observed in the sequence, we have that a majority of those wallets has been generated by honest players using independent randomnesses. Essentially our blockchain assumption builds on top of the standard chain quality

[1] Here we refer to wallets identifying the block leader cashing the reward and not to wallets involved in transactions.

assumption, requiring that the adversary will be the "winning" node that decides the next block less often than honest players, without taking into account blocks that reuse a known wallet. We remark that our blockchain assumption is not implied by chain quality but still consists of considering bounds on the power of the adversary in deciding what happens on the blockchain.

Note that because of forks, a j-th block B generated by a honest player that becomes permanently added to the blockchain (i.e., a stable block) might not be the only j-th block that has been generated by a honest player. We assume that for every block index j there is a known bound $u \in \mathbb{N}$ that limits the number of blocks B proposed by honest players satisfying the predicate and that can be legitimately added as j-th blocks.

We will leverage this blockchain assumption and the simulator's control of the honest majority to build a pvZK proof as follows. The high-level idea is to follow the celebrated FLS approach [FLS90] and prove the OR of two statements: either "x in L" or "Previously I have predicted the majority of fresh wallets appeared in the last t blocks". In particular our idea reminds the implementation of the FLS approach proposed by Barak [Bar01] where the trapdoor theorem consists of some unpredictable information that will be decided by honest players in the future. Indeed the soundness of our construction will follow from similar arguments and will actually be simpler. The reason is that we implement the prediction step with statistically binding commitments and thus, unlike Barak, we will not have to worry about a prover breaking binding or finding collisions in a collision-resistant hash function.

To implement this approach we need two ingredients: a non-interactive statistically binding commitment scheme (that can be constructed from one-way permutations) and a publicly verifiable witness indistinguishable proof system pvWI. We use the pvWI proof system recently constructed in [SSV19] which is the first pvWI proof system from a blockchain assumption. Our blockchain assumption implies the one of [SSV19]. Such proof system leverages the underlying blockchain assumption by providing an interactive prover and a non-interactive verification function. Concretely, the pvWI proof of [SSV19] builds on a 3-round WI proof system where the first two rounds are played between the prover and blockchain: the prover posts the first round of the WI proof on the blockchain, then she waits for a few blocks extending the block containing the first message and from those extracts a challenge. The third round of the WI proof is then sent to the actual verifier, who can use the blockchain to validate all 3 rounds, non-interactively. We need the following 3 properties from the pvWI proof: (1) *delayed-input* completeness, which means that the prover will use the theorem only for computing the last message of the protocol, (2) *adaptive security* (assuming secure-erasure) even *in presence of blockchain failure*, that is, even if the adversary corrupts all blockchain players (and even the prover), the WI property is preserved (assuming the prover has deleted the relevant randomness) and (3) unconditional soundness[2] in presence of Assumption 1. Since such properties were not explicitly claimed in [SSV19] we also make a minor update to their protocol and make explicit these extra features that will be crucial to instantiate the pvWI proof (i.e., a subprotocol) and therefore to realize our main construction.

With the above ingredients in hands, our pvZK proof system works as follows. First, the prover commits to $u{\cdot}t$ strings $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ (where u, t are parameters of our blockchain assumption) and posts the commitments and the first round of the pvWI proof on the blockchain. Then, she waits until the blockchain is extended by a sequence of t blocks $\overline{B}_1, \ldots, \overline{B}_t$, that include n blocks $B_1, \ldots, B_n$ with fresh field values (that is, with values that were not observed before). Let $v_i, \ldots, v_n$ be such fresh values observed on the blockchain. In the final step, the prover computes the last round of the pvWI proof, for the theorem:"$x \in \mathcal{L}$ or $(\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t})$ are commitments of at least $n/2 + 1$ of the values $(v_1, \ldots, v_n)$".

The simulator $\mathsf{S}_{\mathsf{NIZK}}$ uses the same power of the simulator of [GG17] controlling the honest players in the simulated experiment (in particular, the simulator adds the blocks in the blockchain on behalf of honest players). Therefore $\mathsf{S}_{\mathsf{NIZK}}$ can predict the majority of the unpredictable new wallets associated with a sequence of t future blocks, and use this knowledge as a trapdoor theorem when computing the messages of the pvWI proof. Notice that the simulator can not tightly predict the future wallets that will be permanently added to the blockchain since there will be several other honest blocks to simulate that will circulate in the network, might even appear in some forks but eventually will not be part of the blockchain. Since the simulator has no direct power to decide

[2] See the paragraph below about the power of the adversary.

which branch of a fork will remain in the blockchain, we require way more than just t commitments. Indeed we consider the parameter u that measures the upper bound on the amount of valid blocks with fresh wallets that honest players propose for each index of the sequence of blocks of the blockchain.

Clearly even an unbounded malicious prover that does not violate our blockchain assumption can not predict a majority of the future fresh wallets and this argument will guarantee the soundness. Moreover our pvZK proof can be run computing all but the last message before even knowing the statement to prove (i.e., it enjoys delayed-input completeness and adaptive-input zero knowledge and soundness).

On the power of the adversary. In a publicly verifiable proof assuming that an adversarial prover is PPT does not really say much about his limits with respect to the security of the blockchain. Indeed in case of proof-of-work blockchains the limitation of the adversary should be compared to the overall computational capabilities of the network rather than compared to a generic polynomial on input the security parameter. In our definition of soundness we will therefore consider an unbounded adversary. When proving the security of our construction we will state explicitly our blockchain assumption and implicitly we will assume that the constraints on the adversary (see Section 2.2.2) required by the underlying blockchain are maintained.

2.1.2 Related Work

The idea of replacing trusted setup with a blockchain assumption has been explored already for other cryptographic properties that we know how to achieve only using trusted parties. Examples are, fair multi-party computation [BK14a] and random beacon generation [BGZ16a]. A recent work [CGJ19] investigated using the blockchain as a global setup assumptions to construct concurrent self-composable secure computation protocol, which is impossible in the standard model. We stress that [CGJ19] does not provide public verifiability.

2.2 Definitions

Preliminary. We denote the security parameter by λ and use "||" as concatenation operator (i.e., if a and b are two strings then by $a||b$ we denote the concatenation of a and b). We use the abbreviation PPT that stays for probabilistic polynomial time. We use poly to indicate a generic polynomial function. A *polynomial-time relation* $\mathcal{R}$ (or *polynomial relation*, in short) is a subset of $\{0,1\}^* \times \{0,1\}^*$ such that membership of (x,w) in $\mathcal{R}$ can be decided in time polynomial in $|x|$. For $(x,w) \in \mathcal{R}$, we call x the *instance* and w a *witness* for x. For a polynomial-time relation $\mathcal{R}$, we define the $\mathcal{NP}$-language $L_{\mathcal{R}}$ as $L_{\mathcal{R}} = \{x \text{ s.t. } \exists\, w : (x,w) \in \mathcal{R}\}$. Analogously, unless otherwise specified, for an $\mathcal{NP}$-language L we denote by $\mathcal{R}$ the corresponding polynomial-time relation (that is, $\mathcal{R}$ is such that $L = L_{\mathcal{R}}$). We will denote by $\mathcal{P}^{\mathsf{st}}$ a stateful algorithm $\mathcal{P}$ with state st. We will use the notation $r \in_R \{0,1\}^\lambda$ to indicate that r is sampled at random from $\{0,1\}^\lambda$. When we want to specify the randomness r used by an algorithm Al we use the following notation $\mathsf{Al}(\cdot; r)$.

2.2.1 Blockchain Protocols

In the next two sections we borrow the description of a blockchain protocol of [PSS17, GG17], moreover we explicitly define the procedure executed by an honest player in order to add a block. A blockchain protocol Γ is parameterized by a validity predicate V that captures the semantics and rules of the blockchain. Γ consists of 4 polynomial-time algorithms $(\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ with the following syntax.

- $\mathsf{UpdateState}(1^\lambda, \mathsf{st})$: It takes as input the security parameter λ, state st and outputs the updated state st.
- $\mathsf{GetRecords}(1^\lambda, \mathsf{st})$: It takes as input the security parameter λ and state st. It outputs the longest ordered sequence of valid blocks $\mathbf{B}$ (or simply blockchain) contained in the state variable, where each block in the chain itself contains an unordered sequence of records messages.
- $\mathsf{Broadcast}(1^\lambda, m)$: It takes as input the security parameter λ and a message m, and broadcasts the message over the network to all nodes executing the blockchain protocol. It does not give any output.

- $\mathsf{GenBlock}(\mathsf{st}, \mathbf{B}, x)$: It takes as input a state st, a blockchain $\mathbf{B}$[3], $x \in \{0,1\}^*$ and outputs a candidate block B that contains a string v computed running a function f_{ID} that is defined as follows. The function $f_{\mathsf{ID}}(1^\lambda; r)$ takes as input the security parameter λ and running with $\mathsf{poly}(\lambda)$ bits of randomness r outputs a q bit string v, where $q = \mathsf{poly}(\lambda)$. Moreover every time that f_{ID} runs on input a freshly generated randomness it holds that $H_\infty(f_{\mathsf{ID}}(1^\lambda; \cdot)) \geq \lambda$[4]. The generated block B could satisfy or not the validity predicate V.
  We will denote by B_v a block B that contains the string v computed using f_{ID}.

Blockchain notation. With the notation $\mathbf{B} \leq \mathbf{B}'$ we will denote that the blockchain $\mathbf{B}$ is a prefix of the blockchain $\mathbf{B}'$. We denote by $\mathbf{B}^{\lceil n}$ the chain resulting from "pruning" the last n blocks in $\mathbf{B}$. We will denote by Γ^{V} a blockchain protocol Γ that has validate predicate V. A blockchain $\mathbf{B}$ generated by the execution of Γ^{V} is the blockchain obtained by an honest player after calling $\mathsf{GetRecords}$ during an execution of Γ^{V}. An honest execution of $\mathsf{GenBlock}$ is an execution of $\mathsf{GenBlock}$ computed by an honest player. The blockchain protocol Γ^{V} satisfies the property of chain-consistency property, chain-growth and chain-quality defined in previous works [GKL15, PSS17]. In the rest of the chapter we will denote by $\eta(\cdot)$ the chain consistency parameter of Γ^{V}.

Definition 1 (Block Trim Function)**.** Let B_v be a block generated using $\mathsf{GenBlock}$ that satisfies V. We define a block trim function as a deterministic function $\mathtt{trim}$ that on input B_v outputs v.

Note that for two blocks B, B' that satisfy V and are generated by an honest execution of $\mathsf{GenBlock}$ it could happen that $\mathtt{trim}(B) = \mathtt{trim}(B')$ since $\mathsf{GenBlock}$ in the two executions could run f_{ID} on input the same randomness stored in the state of Pt.

Definition 2 (Good Execution of $\mathsf{GenBlock}$)**.** Let $\overline{\mathbf{B}}$ be a blockchain generated by an execution of Γ^{V}. An execution of $\mathsf{GenBlock}$ is good w.r.t. a blockchain $\mathbf{B}$ if it holds that $\mathsf{GenBlock}$ runs on input $\overline{\mathbf{B}}$ s.t. $\overline{\mathbf{B}}^{\lceil \eta_1(\lambda)} \geq \mathbf{B}$, moreover $\mathsf{GenBlock}$ runs f_{ID} on input fresh randomness and outputs a block that satisfies the validity predicate V.

Definition 3 (Good Block)**.** A block produced by an honest player running a good execution of $\mathsf{GenBlock}$ is a *good* block.

Definition 4 (Pristine Block)**.** Let $\mathtt{trim}$ be the block trim function defined in Definition 1. Let $\mathbf{B}$ be a blockchain composed of k blocks generated by an execution of Γ^{V}. The j-th block B_j of $\mathbf{B}$ is *pristine* if for each B_i of $\mathbf{B}$ with $0 < i < j$ it holds that $v \neq v_i$ where $v = \mathtt{trim}(B_j)$ and $v_i = \mathtt{trim}(B_i)$.

Assumption 1. Let $\mathbf{B}$ be a blockchain generated during an execution of Γ^{V}. There exists $t = \mathsf{poly}(\lambda)$ and $u = \mathsf{poly}(\lambda)$ such that for any sequence of t consecutive blocks $B_{i+1}, \ldots, B_{i+t}$ added to $\mathbf{B}$ during the execution of Γ^{V}, let n be the number of pristine blocks in $B_{i+1}, \ldots, B_{i+t}$, it holds that:

1. At least $n/2 + 1$ of the pristine blocks in the sequence $B_{i+1}, \ldots, B_{i+t}$ have been generated by honest players through good executions of $\mathsf{GenBlock}$;
2. For each $j \in \{1, \ldots t\}$, the probability that honest players obtain through honest executions of $\mathsf{GenBlock}$ $u' > u$ different blocks satisfying the validity predicate for the position $i+j$ in the blockchain is negligible.

We refer to t as the *pristine parameter* and to u as the *attempts parameter*.

For the sake of simplifying the description of our construction we will assume wlog that n is also a pristine parameter.

[3] In order to simplify future parts of the Chapter we make an abuse of notation and we explicitly add the blockchain as input of $\mathsf{GenBlock}$ even though the blockchain can be computed running $\mathsf{GetRecords}$ on input st.

[4] In the existing blockchain the value v could be an identifier of a wallet and f_{ID} is the randomized function that generates it.

2.2.2 Execution of Γ^{V} in an Environment

At a very high level, the execution of the protocol Γ^{V} proceeds in rounds that model time steps. Each participant in the protocol runs the UpdateState algorithm to keep track of the current (latest) blockchain state. This corresponds to listening on the broadcast network for messages from other nodes. The GetRecords algorithm is used to extract an ordered sequence of blocks encoded in the blockchain state variable. The Broadcast algorithm is used by a player when she wants to post a new message m on the blockchain. Note that the message m is accepted by the blockchain protocol only if it satisfies the validity predicate V given the current state, (i.e., the current sequence of blocks).

Following prior works [GKL15, KP15, PSS17], we define the protocol execution following the activation model of the Universal Composability framework of [Can01a] (though like [GG17] we will not prove UC-security of our results). For any blockchain protocol $\Gamma^{\mathsf{V}}(\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$, the protocol execution is directed by the environment $\mathcal{Z}(1^\lambda)$. The environment $\mathcal{Z}$ activates the players as either honest or corrupt and is also responsible for providing inputs/records to all players in each round.

All the corrupt players are controlled by the adversary $\mathcal{A}$ that can corrupt them adaptively after that the execution of Γ^{V} started.

Specifically $\mathcal{A}$ can send a corruption request $< \mathtt{corr}, \mathsf{Pt}_\mathtt{i} >$ to player Pt_i at any point during the execution of Γ^{V}. The adversary is also responsible for the delivery of all network messages. Honest players start by executing UpdateState on input 1^λ with an empty state $\mathsf{st} = \epsilon$.

- In round r, each honest player Pt_i potentially receives a message(s) m from $\mathcal{Z}$ and potentially receives incoming network messages (delivered by $\mathcal{A}$). It may then perform any computation, broadcast a message (using Broadcast algorithm) to all other players (which will be delivered by the adversary; see below) and update its state st_i. It could also attempt to "add" a new block to its chain: Pt_i will run the procedure GenBlock, and this execution of GenBlock will be good (see Definition 2) if requested by $\mathcal{Z}$.
- $\mathcal{A}$ is responsible for delivering all messages sent by players (honest or corrupted) to all other players. $\mathcal{A}$ cannot modify the content of messages broadcast by honest players, but it may delay or reorder the delivery of a message as long as it eventually delivers all messages within a certain time limit.
- At any point $\mathcal{Z}$ can communicate with adversary $\mathcal{A}$.

Constraints on the adversary. In order to show that a blockchain enjoys some useful properties (e.g., chain consistency) prior works [PSS17, GKL15] restrict their analysis to compliant executions of Γ^{V} where some specific restrictions[5] are imposed to $\mathcal{Z}$ and $\mathcal{A}$. Those works showed that certain desirable security properties are respected except with negligible probability in any compliant execution. Obviously, when in this chapter we claim results assuming some properties of the blockchain, we are taking into account compliant executions of the underlying blockchain protocol only. The same is done by [GG17].

2.2.3 Publicly Verifiable ZK Proof System from Blockchains

Here we define delayed-input publicly verifiable zero knowledge w.r.t. blockchain failure over a blockchain protocol $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$. We will make use of the following notation.

The view of a player Pt consists of the messages received during an execution of Γ^{V}, along with its randomness and its inputs. Let $\mathsf{Exec}^{\Gamma^{\mathsf{V}}}(\mathcal{A}, \mathcal{H}, \mathcal{Z}, 1^\lambda)$ be the random variable denoting the joint view of all players in the execution Γ^{V}, this joint view view fully determines the execution. Let $\Gamma^{\mathsf{V}}_{\mathsf{view}}(\mathcal{A}, \mathcal{H}, \mathcal{Z}, 1^\lambda)$ denote an execution of $\Gamma^{\mathsf{V}}(\mathcal{A}, \mathcal{H}, \mathcal{Z}, 1^\lambda)$ producing view as joint view.

Definition 5 (Publicly Verifiable Proof System from Blockchain)**.** A pair of stateful PPT algorithms $\Pi = (\mathcal{P}, \mathcal{V})$ over a blockchain protocol Γ^{V} is a publicly verifiable proof system for the $\mathcal{NP}$-language $\mathcal{L}$ with witness relation $\mathcal{R}$ if it satisfies the following properties:

Completeness. $\forall\ x, w$ s.t. $(x, w) \in \mathcal{R}$, $\forall$ PPT adversary $\mathcal{A}$ and set of honest players $\mathcal{H}$ and environment $\mathcal{Z}$, assuming that $\mathcal{P} \in \mathcal{H}$, there exist negligible functions $\nu_1(\cdot)$, $\nu_2(\cdot)$ such that:

[5] For instance, they require that any broadcasted message is delivered in a maximum number of time steps.

$$\Pr\left[\mathcal{V}(x,\pi,\mathbf{B})=1 \;\vdots\; \begin{array}{c}\mathsf{view}\leftarrow \mathsf{Exec}^{\Gamma^{\mathsf{V}}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)\\ \pi\leftarrow \mathcal{P}^{\mathsf{st}_{\mathcal{P}}}(x,w)\\ \mathbf{B}=\mathsf{GetRecords}(1^\lambda,\mathsf{st}_j)\end{array}\right]\geq 1-\nu_1(|x|)-\nu_2(\lambda)$$

where $\mathsf{st}_{\mathcal{P}}$ denotes the state of $\mathcal{P}$ during the execution $\Gamma^{\mathsf{V}}_{\mathsf{view}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$. The running time of $\mathcal{P}$ is polynomial in the size of the blockchain $\mathbf{B}=\mathsf{GetRecords}(1^\lambda,\mathsf{st}_j)$ where st_j is the state of $\mathsf{Pt}_j\in\mathcal{H}$ at the end of the execution $\Gamma^{\mathsf{V}}_{\mathsf{view}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$.[6]

Soundness. $\forall\ x\notin\mathcal{L}$, $\forall$ stateful adversary $\mathcal{A}$ and set of honest players $\mathcal{H}$ and environment $\mathcal{Z}$, there exist negligible functions $\nu_1(\cdot)$, $\nu_2(\cdot)$ such that:

$$\Pr\left[\mathcal{V}(x,\pi,\mathbf{B})=1 \;\vdots\; \begin{array}{c}\mathsf{view}\leftarrow \mathsf{Exec}^{\Gamma^{\mathsf{V}}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)\\ \pi,x\leftarrow \mathcal{A}^{\mathsf{st}_{\mathcal{A}}}\\ \mathbf{B}=\mathsf{GetRecords}(1^\lambda,\mathsf{st}_j)\end{array}\right]\leq \nu_1(|x|)+\nu_2(\lambda)$$

where $\mathsf{st}_{\mathcal{A}}$ denotes the state of $\mathcal{A}$ during the execution $\Gamma^{\mathsf{V}}_{\mathsf{view}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$. Furthermore st_j is the state of an honest player $\mathsf{Pt}_j\in\mathcal{H}$ at the end of the execution $\Gamma^{\mathsf{V}}{}_{\mathsf{view}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$.

The proof π might consist of multiple messages, i.e., $\pi=(\pi^1,\ldots,\pi^m)$, in this case, we will say that Π is an m-messages proof system. Moreover if π is composed of m-messages $\pi=(\pi^1,\ldots,\pi^m)$, $\mathcal{A}$ is allowed to choose x just before computing the last message π^m of the proof $\pi=(\pi^1,\ldots,\pi^m)$.

If the soundness holds only against PPT adversary $\mathcal{A}$, then we say that Π is an *argument system*, instead of a proof system.

Definition 6 (Delayed-Input Completeness from Blockchain)**.** An m-messages Π proof system over a blockchain protocol Γ^{V} is **delayed-input**, if Π satisfies completeness when x,w are involved only in the computation of last message π^m of the proof $\pi=(\pi^1,\ldots,\pi^m)$.

Definition 7 (Witness Indistinguishability w.r.t. Blockchain Failure)**.** A publicly verifiable proof system $\Pi=(\mathcal{P},\mathcal{V})$ over a blockchain protocol Γ^{V} for the $\mathcal{NP}$-language $\mathcal{L}$ with witness relation $\mathcal{R}$ is witness indistinguishable (WI) w.r.t. blockchain failure if it satisfies the following properties.

Let $\mathsf{st}_{\mathsf{Pt}_i}$ denote the state of a player Pt_i in a execution of the blockchain protocol Γ^{V}. $\forall\ x,w_0,w_1$ such that $(x,w_0)\in\mathcal{R}$ and $(x,w_1)\in\mathcal{R}$, $\forall$ PPT adversary $\mathcal{A}$ and set of honest players $\mathcal{H}$ and environment $\mathcal{Z}$, where $\mathcal{P}\in\mathcal{H}$ and for any $b\in\{0,1\}$ it holds that:

$$\left\{\mathsf{view}_{\mathcal{A}}:\mathsf{view}_{\mathcal{A}}\leftarrow \mathtt{Exp}^{0}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda,x,w_0)\right\}\approx\left\{\mathsf{view}_{\mathcal{A}}:\mathsf{view}_{\mathcal{A}}\leftarrow \mathtt{Exp}^{1}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda,x,w_1)\right\}$$

where $\mathtt{Exp}^{b}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda,x,w_b)$ is defined below, for $b\in\{0,1\}$.

$\mathtt{Exp}^{b}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda,x,w_b)$:
- $\mathcal{P}$ runs on input (x,w_b). - An execution of $\Gamma^{\mathsf{V}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$ starts. -$\mathcal{P}^{\mathsf{st}_{\mathcal{P}}}$outputs π, where $\mathsf{st}_{\mathcal{P}}$ is the state of $\mathcal{P}$ in the execution $\Gamma^{\mathsf{V}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$. -$\mathcal{A}$ can send a collapse request $<\mathtt{corr},\mathtt{all}>$ obtaining $\mathsf{st}_{\mathcal{P}}$ from $\mathcal{P}$ and $\mathsf{st}_1,\ldots,\mathsf{st}_{\lvert\mathcal{H}\rvert}$where st_i is the state of $\mathsf{Pt}_i\in\mathcal{H}$. -The execution of $\Gamma^{\mathsf{V}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$ terminates producing view. -$\mathcal{A}$ outputs her view $\mathsf{view}_{\mathcal{A}}$ and this is the output of the experiment.

[6]Note that the execution of $\mathsf{Exec}^{\Gamma^{\mathsf{V}}}(\mathcal{A},\mathcal{H},\mathcal{Z},1^\lambda)$ could continue even after π is provided by $\mathcal{P}$.

Moreover, if π is composed of multiple messages $\pi = (\pi^1, \ldots, \pi^m)$ $\mathcal{A}$ is allowed to choose x any time before that the last message π^m of $\pi = (\pi^1, \ldots, \pi^m)$ is computed.

Definition 8 (Zero-Knowledge w.r.t. Blockchain Failure). A publicly verifiable proof system $\Pi = (\mathcal{P}, \mathcal{V})$ over a blockchain protocol Γ^{V} for the $\mathcal{NP}$-language $\mathcal{L}$ with witness relation $\mathcal{R}$ is Zero-Knowledge w.r.t. blockchain failure (pvZK) w.r.t blockchain failure if it satisfies the following property:

Let $\mathsf{st}_{\mathsf{Pt}_i}$ denoting the state of a player Pt_i in execution of the blockchain protocol Γ^{V}. There is a stateful PPT algorithm S such that $\forall\ x, w$ s.t. $(x, w) \in \mathcal{R}$, $\forall$ PPT adversary $\mathcal{A}$ and set of honest players $\mathcal{H}$ and environment $\mathcal{Z}$, where $\mathcal{P} \in \mathcal{H}$ it holds that:

$$\left\{\mathsf{view}_{\mathcal{A}} : \mathsf{view}_{\mathcal{A}} \leftarrow \mathtt{Exp}^{0}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda, w, x)\right\} \approx \left\{\mathsf{view}_{\mathcal{A}} : \mathsf{view}_{\mathcal{A}} \leftarrow \mathtt{Exp}^{1}{}_{\mathcal{A},\Pi,\mathsf{S},\Gamma^{\mathsf{V}}}(\lambda, x)\right\}$$

where $\mathtt{Exp}^{0}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda, x, w)$ and $\mathtt{Exp}^{1}{}_{\mathcal{A},\Pi,\mathsf{S},\Gamma^{\mathsf{V}}}(\lambda, x)$ are defined below.

$\mathtt{Exp}^{0}{}_{\mathcal{A},\Pi,\Gamma^{\mathsf{V}}}(\lambda, x, w)$:

- $\mathcal{P}$ runs on input (x, w).
- An execution of $\Gamma^{\mathsf{V}}(\mathcal{A}, \mathcal{H}, \mathcal{Z}, 1^\lambda)$ starts.
  1. At any point $\mathcal{A}$ can send a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ to $\mathcal{P}$ obtaining $\mathsf{st}_{\mathcal{P}}$, (where $(x, \mathtt{w}) \in \mathcal{R}$).
  2. $\mathcal{P}^{\mathsf{st}_{\mathcal{P}}}$ outputs π.
  3. $\mathcal{A}$ can send a collapse request $< \mathtt{corr}, \mathtt{all} >$ obtaining:
     $\mathsf{st}_1, \ldots, \mathsf{st}_{|\mathcal{H}|}$ where st_i is the state of $\mathsf{Pt}_i \in \mathcal{H}$;
     $\mathsf{st}_{\mathcal{P}}$ if $\mathcal{A}$ did not compute Step 1.

-The execution of $\Gamma^{\mathsf{V}}(\mathcal{A}, \mathcal{H}, \mathcal{Z}, 1^\lambda)$ ends producing view.
-$\mathcal{A}$ outputs her view $\mathsf{view}_{\mathcal{A}}$ in view and this is the output of the experiment.

$\mathtt{Exp}^{1}{}_{\mathcal{A},\Pi,\mathsf{S},\Gamma^{\mathsf{V}}}(\lambda, x)$:

- S runs on input x.
- An execution of $\Gamma^{\mathsf{V}}(\mathcal{A}, \mathsf{S}, \mathcal{Z}, 1^\lambda)$ starts.
  1. At any point $\mathcal{A}$ can send a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ to S:
     S provides the state $\mathsf{st}_{\mathcal{P}}$ of an honest prover of Π, (where $(x, w) \in \mathcal{R}$).
  2. S outputs π.
  3. $\mathcal{A}$ can send a collapse request $< \mathtt{corr}, \mathtt{all} >$:
     S provides $\mathsf{st}_1, \ldots, \mathsf{st}_{|\mathcal{H}|}$ where st_i is the state of $\mathsf{Pt}_i \in \mathcal{H}$;
     S provides $\mathsf{st}_{\mathcal{P}}$ if $\mathcal{A}$ did not compute Step 1.

-The execution of $\Gamma^{\mathsf{V}}(\mathcal{A}, \mathsf{S}, \mathcal{Z}, 1^\lambda)$ ends producing view.
-$\mathcal{A}$ outputs her view $\mathsf{view}_{\mathcal{A}}$ in view and this is the output of the experiment.

2.3 Publicly Verifiable ZK Proof System

We construct a delayed-input publicly verifiable zero-knowledge proof system w.r.t. blockchain failure $\Pi_{\mathsf{NIZK}} = (\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ over any blockchain protocol $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ satisfying Assumption 1. The parameters of Π_{NIZK} are reported in Table 2.1, we recall that for easy of exposition we consider that wlog in a sequence of t blocks, n of them are pristine, where n is an even integer.
Π_{NIZK} for the $\mathcal{NP}$-language $\mathcal{L}$ makes use of the following tools:

Table of Notation	
ℓ	size of the theorem for $\mathcal{L}_{\mathsf{pvWI}}$.
m	number of messages of Π_{pvWI}.
q	output-length of the block trim function $\mathtt{trim}$. See Definition 1.
η	chain consistency parameter of Γ^{V}.
t, n	pristine parameters of Γ^{V}. See Assumption 1.
u	attempts parameter of Γ^{V}. See Assumption 1.

Table 2.1: Parameters of Π_{NIZK}.

- The block trim function $\mathtt{trim}$ defined in Definition 1, that on input a block B outputs a q-bits long string v.
- A non-interactive statistically binding commitment scheme $\Pi_{\mathsf{Com}} = (\mathsf{Com}, \mathsf{VrfyOpen})$.
- A delayed-input publicly verifiable WI proof system w.r.t. blockchain failure $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$ over any blockchain protocol $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ for $\mathcal{NP}$-language $\mathcal{L}_{\mathsf{pvWI}}$ associated to the relation $\mathcal{R}_{\mathsf{pvWI}} = \big\{((x, x_{\mathtt{com}}), w) : (x, w) \in \mathcal{R}) \ \vee \ (x_{\mathtt{com}}, w) \in \mathcal{R}_{\mathtt{com}} \big\}$, where $\mathcal{R}$ is the relation associated to the $\mathcal{NP}$-language $\mathcal{L}$ and $\mathcal{R}_{\mathtt{com}}$ is the relation associated to the following $\mathcal{NP}$-language:

$$\mathcal{L}_{\mathtt{com}} = \Big\{ \{\mathtt{com}_j\}_{j=1}^{u\cdot t}, \{v_i\}_{i=1}^{n} : \exists\ 1 \le j_1 < \cdots < j_{n/2+1} \le n,\ \{\mathsf{Dec}_{j_k}\}_{k=1}^{n/2+1}$$
$$\text{s.t. } \mathsf{VrfyOpen}(\mathtt{com}_{j_k}, \mathsf{Dec}_{j_k}, v_{j_k}) = 1\ \forall k = 1, \ldots, n/2+1 \Big\}$$

  Loosely speaking the relation $\mathcal{R}_{\mathtt{com}}$ is satisfied if the message committed in $\mathtt{com}_{j_k}$ is v_{j_k} for at least $n/2+1$ distinct values of j_k. The instance length of $\mathcal{L}_{\mathsf{pvWI}}$ is ℓ and the size of the proof generated by $\mathcal{P}_{\mathsf{pvWI}}$ is of m messages.

Our delayed-input publicly verifiable zero-knowledge proof system w.r.t. blockchain failure $\Pi_{\mathsf{NIZK}} = (\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ is described in Figure 2.1.

Theorem 1. *Let $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ be a any blockchain protocol that satisfies Assumption 1. Let $\Pi_{\mathsf{Com}} = (\mathsf{Com}, \mathsf{VrfyOpen})$ be a non-interactive statistically binding commitment scheme. Let $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$ be a delayed-input public verifiable WI w.r.t. blockchain failure proof system over Γ^{V} for $\mathcal{NP}$-language $\mathcal{L}_{\mathsf{pvWI}}$. Assuming erasure, $\Pi_{\mathsf{NIZK}} = (\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ (described in Figure 2.1) is a delayed-input publicly verifiable zero-knowledge proof system w.r.t. blockchain failure over Γ^{V} for $\mathcal{NP}$.*

We note that a pvWI proof that satisfies delayed-input completeness can be instantiated from OWPs using the work of [SSV19]. In Section 2.5 we prove that Π_{pvWI} satisfies Definitions 5, 7. Therefore we have the following corollary.

Corollary 1. *Let $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ be a blockchain protocol that satisfies Assumption 1. Assuming erasure, if one-way permutations exists, then $\Pi_{\mathsf{NIZK}} = (\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$ is a delayed-input publicly verifiable zero knowledge proof system w.r.t. blockchain failure over Γ^{V} for $\mathcal{NP}$.*

The proof of the Theorem 1 and the description of the simulator $\mathsf{S}_{\mathsf{NIZK}}$ for Π_{NIZK} can be found in the next subsections.

Note that the inputs of Π_{NIZK} (i.e. the statement x and the witness w) are used only in the last message of the protocol. This means that the prover can *pre-process* the first messages ahead of time (even without knowing the statement) and complete the last message whenever the statement becomes available.

2.3.1 Delayed-Input Completeness (Definition 6)

Let st and $\mathsf{st}_{\mathsf{Pt}_i}$ be respectively the states of $\mathcal{P}$ and of an honest player Pt_i after Step 6 of Π_{NIZK} (that is, after the proof has been computed). Since both $\mathcal{P}$ and $\mathcal{V}$ are running the protocol honestly, from the chain-

PUBLICLY VERIFIABLE ZK PROOF SYSTEM W.R.T. BLOCKCHAIN FAILURE $\Pi_{\mathsf{NIZK}} = (\mathcal{P}_{\mathsf{NIZK}}, \mathcal{V}_{\mathsf{NIZK}})$

Parameters are defined in Table 2.1.

PROVER PROCEDURE: $\mathcal{P}_{\mathsf{NIZK}}$. Input: instance x, witness w s.t. $(x, w) \in \mathcal{R}$.

— **First step.**

1. Compute $(\mathtt{com}_j, \mathsf{Dec}_j) \leftarrow \mathsf{Com}(0^q)$ and erase Dec_j for $j = 1, \ldots, t \cdot u$.

— **Blockchain Interaction.**

2. Set $\mathsf{st} = \epsilon$. Post $\mathtt{com}_1, \ldots, \mathtt{com}_{u \cdot t}$ on the blockchain by running $\mathsf{Broadcast}(1^\lambda, \mathtt{com}_1, \ldots, \mathtt{com}_{u \cdot t})$ and then monitor the blockchain by running $\mathsf{st} = \mathsf{UpdateState}(1^\lambda, \mathsf{st})$, $\mathbf{B} = \mathsf{GetRecords}(1^\lambda, \mathsf{st})$, until $\mathtt{com}_1, \ldots, \mathtt{com}_{u \cdot t}$ followed by t additional blocks $B_1, \ldots, B_t$ are posted on the blockchain $\mathbf{B}^{\lceil \eta}$. Let $\overline{B_1}, \ldots, \overline{B_n}$ be the n pristine blocks in the sequence $B_1, \ldots, B_t$.

— **Second step.**

3. Compute $v_j = \mathtt{trim}(\overline{B_j})$ for $j = 1, \ldots, n$ and set $\mathtt{com} = \{\mathtt{com}_j\}_{j=1}^{u \cdot t}$, $\mathtt{val} = \{v_j\}_{j=1}^{n}$, $x_{\mathtt{com}} = (\mathtt{com}, \mathtt{val})$, $x_{\mathsf{pvWI}} = (x, x_{\mathtt{com}})$.
4. Obtain π^1_{pvWI} with randomness r_1 executing $\mathcal{P}_{\mathsf{pvWI}}$ on input $1^\lambda, \ell$ and interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
5. For $i = 2, \ldots, m-1$:
   Obtain π^i_{pvWI} with randomness r_i executing $\mathcal{P}_{\mathsf{pvWI}}$ on input $\pi^{i-1}_{\mathsf{pvWI}}$ and interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
6. Obtain π^m_{pvWI} executing $\mathcal{P}_{\mathsf{pvWI}}$ on input $\pi^{m-1}_{\mathsf{pvWI}}, x_{\mathsf{pvWI}}, w$ and interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
7. Set $\pi_{\mathsf{pvWI}} = (\pi^1_{\mathsf{pvWI}}, \ldots, \pi^m_{\mathsf{pvWI}})$ and $\pi = (x_{\mathsf{pvWI}}, \{\mathtt{com}_j\}_{j=1}^{u \cdot t}, \pi_{\mathsf{pvWI}})$ erase any coins that $\mathcal{P}_{\mathsf{pvWI}}$ requests to erase and output π.

VERIFIER PROCEDURE: $\mathcal{V}_{\mathsf{NIZK}}$. Input: x, $\pi = (x_{\mathsf{pvWI}}, \{\mathtt{com}_j\}_{j=1}^{u \cdot t}, \pi_{\mathsf{pvWI}})$, and a blockchain $\tilde{\mathbf{B}}$ works as follows.

— **Check Blockchain.** If the messages $\{\mathtt{com}_j\}_{j=1}^{u \cdot t}$ are not posted on the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ then $\mathcal{V}_{\mathsf{NIZK}}$ outputs 0. Otherwise, let B^* be the block of the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ where the messages $\{\mathtt{com}_j\}_{j=1}^{u \cdot t}$ are posted. Let $\overline{B_1}, \ldots, \overline{B_n}$ be the n pristine blocks of the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ after B^*. $\mathcal{V}_{\mathsf{NIZK}}$ computes $v'_j = \mathtt{trim}(\overline{B_j})$ for $j = 1, \ldots, n$ and parses x_{pvWI} as instance x, commitments $\{\mathtt{com}_j\}_{j=1}^{u \cdot t}$, and strings $\{v_j\}_{j=1}^{n}$.

— **Check Proof.** Accept if all the following conditions are satisfied.
- $v'_j = v_j$ for all $j \in \{1, \ldots, n\}$.
- $\mathcal{V}_{\mathsf{pvWI}}(x_{\mathsf{pvWI}}, \pi_{\mathsf{pvWI}}, \tilde{\mathbf{B}}) = 1$.

EXECUTION OF Γ^{V} BY HONEST PLAYER Pt_j:

Pt_j acts as described in Section 2.2.2, in particular, upon receiving a request of a good execution (see Definition 2) of GenBlock by $\mathcal{Z}$:

Pt_j picks r at random from $\{0,1\}^{\mathsf{poly}(\lambda)}$;

Pt_j runs a good execution of GenBlock and uses the randomness r to execute f_{ID}.

If $\mathcal{A}$ sends a collapse request $< \mathtt{corr}, \mathtt{all} >$, $\mathcal{A}$ obtains $\mathsf{st}_{\mathsf{Pt}_i}$ from honest player Pt_i, for all $i = 1, \ldots, |\mathcal{H}|$, moreover $\mathcal{A}$ obtains the state $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$ of $\mathcal{P}_{\mathsf{NIZK}}$ (if $\mathcal{A}$ did not send a corruption request to $\mathcal{P}_{\mathsf{NIZK}}$ before).

Figure 2.1: Description of our pvZK proof system w.r.t. blockchain failure Π_{NIZK}.

consistency property follows that $\mathbf{B}^{\lceil\eta} \leq \tilde{\mathbf{B}}$ (with overwhelming probability), where $\mathbf{B} = \mathsf{GetRecords}(\mathsf{st})$ and $\tilde{\mathbf{B}} = \mathsf{GetRecords}(\mathsf{st}_{\mathsf{Pt}_i})$. Therefore $\mathcal{V}$ performs all the blockchain checks on $\tilde{\mathbf{B}}$ successfully. After that $\mathcal{P}$ posts the commitments $\{\mathtt{com}_j\}_{j=1}^{u\cdot t}$ in the blockchain $\mathbf{B}$ we are guaranteed by the chain growth property of Γ^{V} and by Assumption 1 that new t blocks will be added to $\mathbf{B}$ and among them n will be pristine. Therefore $\mathcal{P}$ can construct the instance $x_{\mathtt{com}}$ (as defined in Step 3 of Figure 2.1) in order to complete her execution running Π_{pvWI}.

Finally the completeness of Π_{NIZK} follows from the completeness of Π_{pvWI} and the correctness of Π_{Com}.

2.3.2 Soundness (Definition 5)

Claim 1. *Assume that the commitment scheme $\Pi_{\mathtt{com}}$ is statistically binding, Π_{pvWI} is sound and Assumption 1 holds for Γ^{V} then Π_{NIZK} is sound.*

Proof. Let $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ be a successful adversary. Recall that $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ is successful if it produces with non-negligible probability an accepting π of Π_{NIZK} w.r.t. $x \notin \mathcal{L}$, where x is adaptively chosen by $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ before the last message of π.

Let B^* be the block in the blockchain $\mathbf{B}$ where the last commitment of the set of the commitments $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ is posted by $\mathcal{P}^{\star}_{\mathsf{NIZK}}$, and let $B_1, \ldots, B_n$ be the n pristine blocks (in a sequence of t blocks) appeared in $\mathbf{B}$ after the block B^*.

From Assumption 1 it follows that in a sequence of n pristine blocks $B_1, \ldots, B_n$ at least $n/2+1$ are good (see Definition 3). Let $\overline{B}_1, \ldots, \overline{B}_{n/2+1}$ be the $n/2+1$ good blocks in the sequence of pristine blocks $B_1, \ldots, B_n$, and the value $\overline{v}_j$ is s.t. $\overline{v}_j = \mathtt{trim}(\overline{B}_j)$, for $j = 1, \ldots, n/2+1$. When $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ posts $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$, it has no information about the values $\overline{v}_1, \ldots, \overline{v}_{n/2+1}$, because when $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ posts $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ each value $\overline{v}_j$ (for $j = 1, \ldots, n/2+1$) can be guessed with probability $2^{-\lambda}$ (since by Assumption 1 each $\overline{v}_j$ has at least λ bits of min-entropy). Moreover, since Π_{Com} is a perfectly binding commitment scheme, the committed message is uniquely identified in the commitment phase. Therefore the probability that $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ correctly commits the values $\overline{v}_1, \ldots, \overline{v}_{n/2+1}$ is negligible. It follows that the values $\overline{v}_1, \ldots, \overline{v}_{n/2+1}$ are committed in $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ only with negligible probability, therefore $x_{\mathtt{com}} \notin \mathcal{L}_{\mathtt{com}}$. Since by contradiction we are assuming that $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ is successful w.r.t. $x \notin \mathcal{L}$, it follows that with non-negligible probability $x_{\mathsf{pvWI}} = (x_{\mathtt{com}}, x) \notin \mathcal{L}_{\mathsf{pvWI}}$. From the soundness property of Π_{pvWI} follows that $\mathcal{P}^{\star}_{\mathsf{NIZK}}$ is successful only with negligible probability. □

2.3.3 Zero Knowledge w.r.t. Blockchain Failure (Definition 8)

Simulator $\mathsf{S}_{\mathsf{NIZK}}$. The simulator $\mathsf{S}_{\mathsf{NIZK}}$ is presented in Figure 2.2.

Zero Knowledge w.r.t. Blockchain Failure. Let $\mathcal{A}$ be the adversary as defined in Definitions 8. Intuitively, we want to prove that even if the blockchain collapses, the zero-knowledge property of Π_{NIZK} is still preserved.

In order to show that Π_{NIZK} satisfies zero knowledge w.r.t. blockchain failure we will consider the following hybrid experiments.

- Hybrid H_0. In hybrid experiment $H_0(\lambda)$ the simulator $\mathsf{S}'_{\mathsf{NIZK}}$ follows the honest prover procedure of $\mathcal{P}_{\mathsf{NIZK}}$. This experiment is identical to the setting where all proofs are computed according to the honest prover procedure.

- Hybrid H_1. Experiment $H_1(\lambda)$ is described as $H_0(\lambda)$ except that the simulator $\mathsf{S}'_{\mathsf{NIZK}}$ emulates the honest player in the execution of Γ^{V} except that it follows Step 3 and Steps 14-21 of Figure 2.2.

  Note that in $H_0(\lambda)$, after that the commitments are posted in the blockchain, an honest player $\mathsf{Pt}_j \in \mathcal{H}$ upon receiving a request of a good execution (see Definition 2) of $\mathsf{GenBlock}$ from $\mathcal{Z}$ runs f_{ID} on input freshly generated randomness obtaining v. In $H_1(\lambda)$ the value v is generated in the same way as $\mathsf{Pt}_j \in \mathcal{H}$ does in $H_0(\lambda)$ except that v is computed on staring of Π_{NIZK}. Since 1) the values $v_j \leftarrow f_{\mathsf{ID}}(1^{\lambda}; r_j)$ for $j = 1, \ldots, t \cdot u$ are identically distributed in the two hybrid experiments; 2)$\mathsf{S}'_{\mathsf{NIZK}}$ is behaving in the same

SIMULATOR PROCEDURE: $\mathsf{S}_{\mathsf{NIZK}}$.

Parameters are defined in Table 2.1. instance x.

— **First step.**

1. If a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ is received, then execute the steps of $\mathcal{P}_{\mathsf{NIZK}}$ on input x, w. Else continue with the following steps.
2. For $j = 1, \ldots, u \cdot t$:
3. Pick r_j at random from $\{0,1\}^{\mathsf{poly}(\lambda)}$ compute $v_j \leftarrow f_{\mathsf{ID}}(1^\lambda; r_j)$ and set $R = R||r_j$.
4. Compute $(\mathtt{com}_j, \mathsf{Dec}_j) \leftarrow \mathsf{Com}(v_j)$.

— **Blockchain Interaction.**

5. Set $\mathsf{st} = \epsilon$. Post $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ on the blockchain by running $\mathsf{Broadcast}(1^\lambda, \mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t})$ and then monitor the blockchain by running $\mathsf{st} = \mathsf{UpdateState}(1^\lambda, \mathsf{st})$, $\mathbf{B} = \mathsf{GetRecords}(1^\lambda, \mathsf{st})$, until $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ followed by t additional blocks $B_1, \ldots, B_t$ are posted on the blockchain $\mathbf{B}^{\lceil \eta}$. Let $\overline{B_1}, \ldots, \overline{B_n}$ be the n pristine blocks in the sequence $B_1, \ldots, B_t$.

— **Second step.**

6. Compute $v_j = \mathtt{trim}(\overline{B_j})$ for $j = 1, \ldots, n$ and set $\mathtt{com} = \{\mathtt{com}_j\}_{j=1}^{u\cdot t}$, $\mathtt{val} = \{v_j\}_{j=1}^{n}$, $x_{\mathtt{com}} = (\mathtt{com}, \mathtt{val})$, $\pi^0_{\mathsf{pvWI}} = (1^\lambda, \ell)$.
7. Let $B_{j_1}, \ldots, B_{j_k}$ the pristine blocks generated by honest players in the sequence $B_1, \ldots, B_t$ set $w_{\mathtt{com}} = \mathsf{Dec}_{j_1}, \ldots, \mathsf{Dec}_{j_k}$.
8. Obtain π^1_{pvWI} with randomness r_1 executing $\mathcal{P}_{\mathsf{pvWI}}$ on input $1^\lambda, \ell$ and interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
9. For $i = 2, \ldots, m-1$:
   Obtain $r^i, \pi^i_{\mathsf{pvWI}}$ executing $\mathcal{P}_{\mathsf{pvWI}}$ on input r^{i-1}, and $\pi^{i-1}_{\mathsf{pvWI}}$ interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
   If a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ is received.
   Erase the values $\{\mathsf{Dec}_j\}_{j=1}^{u\cdot t}$.
   Output $r' = r'||r^i$ and $\pi^1, \ldots, \pi^i$.
10. If a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ was not received. as in Step 8 to compute π^m.
11. Upon receiving x from $\mathcal{A}$, set $x_{\mathsf{pvWI}} = (x, x_{\mathtt{com}})$
    Obtain π^m_{pvWI} executing $\mathcal{P}_{\mathsf{pvWI}}$ with randomness r^m on input $\pi^{m-1}_{\mathsf{pvWI}}, x_{\mathsf{pvWI}}, w_{\mathtt{com}}$ and interacting with the blockchain if it is required by $\mathcal{P}_{\mathsf{pvWI}}$.
12. Set $\pi_{\mathsf{pvWI}} = (\pi^1_{\mathsf{pvWI}}, \ldots, \pi^m_{\mathsf{pvWI}})$ and $\pi = (x_{\mathsf{pvWI}}, \{\mathtt{com}_j\}_{j=1}^{n}, \pi_{\mathsf{pvWI}})$.
    Output π and in the case a corruption request was not received erase any coins that $\mathcal{P}_{\mathsf{pvWI}}$ requests to erase, moreover erase $\{\mathsf{Dec}\}_{i=1}^{u\cdot t}$.

— EXECUTION OF Γ^{V} SIMULATING HONEST PLAYER Pt_j. Act on behalf of Pt_j as described in Section 2.2.2, in particular, upon receiving a request of a good execution (see Definition 2) of GenBlock by $\mathcal{Z}$:

13. Run $\mathbf{B} = \mathsf{GetRecords}(1^\lambda, \mathsf{st}_j)$, let np be the number of pristine blocks posted after $\mathtt{com}_1, \ldots, \mathtt{com}_{u\cdot t}$ in the blockchain $\mathbf{B}^{\lceil \eta}$. Let K be the number of the blocks added in the blockchain $\mathbf{B}^{\lceil \eta}$. Let nb be the number of honest execution of GenBlock already execute for the block B_{K+1}.
14. If $0 \leq np < n$:
15. Parse R as $r_1, \ldots, r_{u\cdot t}$.
16. Run a good execution of GenBlock on behalf of honest player Pt_j and use the randomness r_{np+nb} to execute f_{ID}.
17. Else:
18. Pick r at random from $\{0,1\}^{\mathsf{poly}(\lambda)}$.
19. Run a good execution of GenBlock on behalf of honest player Pt_j and use the
20. randomness r to execute f_{ID}.
21. If $\mathcal{A}$ sends a collapse request $< \mathtt{corr}, \mathtt{all} >$, $\mathcal{A}$ compute the following steps:
    Disclose state $\mathsf{st}_{\mathsf{Pt}_i}$ from honest player Pt_i, for all $i = 1, \ldots, |\mathcal{H}|$.
    If a corruption request $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ did not occur obtain $\mathsf{st}_{\mathsf{pvWI}}$ from $\mathcal{P}_{\mathsf{pvWI}}$ set $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}} = \mathsf{st}_{\mathsf{pvWI}}$ disclose $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$.

Figure 2.2: Simulator $\mathsf{S}_{\mathsf{NIZK}}$ of Π_{NIZK}.

way of the honest players in an execution of Γ^{V}, we have that $v_1, \ldots, v_{t \cdot u}$ are computed all together), therefore $H_1 \equiv H_0$.

- Hybrid H_2. If a corruption of the form $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ occurs when Π_{NIZK} starts, $H_2(\lambda)$ corresponds to $H_1(\lambda)$, otherwise we consider a series of hybrid experiments $H_2^0(\lambda), \ldots, H_2^{u \cdot t}(\lambda)$ where $H_2^0(\lambda) = H_1(\lambda)$ and $H_2(\lambda) = H_2^{u \cdot t}(\lambda)$ and they are described as follow.

  Hybrid H_2^k with $k \in \{1, \ldots, u \cdot t\}$. The hybrid experiment H_2^k is describe ad H_2^{k-1} except that $\mathsf{S}'_{\mathsf{NIZK}}$ computes the k-th commitment following Steps 2-4 of Figure 2.2. Indeed, $\mathsf{S}'_{\mathsf{NIZK}}$ computes $(\mathtt{com}_j, \mathsf{Dec}_j) \leftarrow \mathsf{Com}(v_j)$ for $j = 1, \ldots, k$ (where $v_j \leftarrow f_{\mathsf{ID}}(1^\lambda; r_j)$) and it computes $(\mathtt{com}_j, \mathsf{Dec}_j) \leftarrow \mathsf{Com}(0^q)$ for $j = k+1, \ldots, u \cdot t$.
  Assuming secure erasure, from Claim 2 it holds that $H_2^{k-1} \approx H_2^k$ for all $k = 1, \ldots, u \cdot t$, therefore since H_1 corresponds to H_2^0 and H_2 corresponds to $H_2^{u \cdot t}$ we conclude that $H_1(\lambda) \approx H_2(\lambda)$.

- Hybrid H_3. If a corruption of the form $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ occurs during the computation of the first $m-1$ messages of Π_{pvWI}, we have that $H_2(\lambda)$ corresponds to $H_3(\lambda)$. Indeed due to the delayed-input property of Π_{pvWI}, $\mathsf{S}_{\mathsf{NIZK}}$ computes the first $m-1$ messages of Π_{pvWI} as $\mathcal{P}_{\mathsf{NIZK}}$ does. Note that the decommitment information $\{\mathsf{Dec}_j\}_{j=1}^{u \cdot t}$ is securely erased by $\mathcal{P}_{\mathsf{NIZK}}$, therefore if $\mathsf{S}_{\mathsf{NIZK}}$ receives a corruption request during the computation π she is able to exhibit random coins that are identically distributed to the one that $\mathcal{P}_{\mathsf{NIZK}}$ would have in her state.

  If a corruption of the form $< \mathtt{ZK}_{\mathtt{corr}}(x, w) >$ does not occur during the computation of the first $m-1$ messages of Π_{pvWI}, then H_3 is defined as follow.

  The hybrid experiment $H_3(\lambda)$ is described exactly as $H_2(\lambda)$ except for the witness used to compute the proof the last message π^m_{pvWI} generated using Π_{pvWI}, for which $\mathsf{S}'_{\mathsf{NIZK}}$ is acting as $\mathsf{S}_{\mathsf{NIZK}}$. In more details, for the computation of the proof π_{pvWI} $\mathsf{S}'_{\mathsf{NIZK}}$ is behaving as described in Steps 11 of Figure 2.2. Assuming secure erasure, since Π_{pvWI} satisfies WI w.r.t. blockchain failure it follows that $H_2(\lambda) \approx H_3(\lambda)$ (see Claim 3).otherwise the two hybrid experiments are

$H_0(\lambda)$ corresponds to the experiment where $\mathcal{P}_{\mathsf{NIZK}}$ is interacting with $\mathcal{A}$ and $H_3(\lambda)$ corresponds to the experiment where $\mathsf{S}_{\mathsf{NIZK}}$ is interacting with $\mathcal{A}$. Since $H_3(\lambda) \approx H_0(\lambda)$ it follows that $\mathcal{A}$ distinguishes the two experiments only with negligible probability.

Claim 2. *Assume that $\Pi_{\mathtt{com}}$ satisfies computationally hiding, secure erasure, and the blockchain protocol Γ^{V} satisfies Assumption 1, then for every pair of messages $m_0, m_1 \in \{0,1\}^q$ it holds that $H_2^{k-1}(\lambda) \approx H_2^k(\lambda)$ for $k \in \{1, \ldots, u \cdot t\}$.*

Proof. Suppose by contradiction that the above claim does not hold, this implies that there exists an adversary $\mathcal{A}$ that is able to distinguish between $H_2^{k-1}(\lambda)$ and $H_2^k(\lambda)$. Note that $\mathcal{A}$ could wait until the protocol Π_{NIZK} ends and then can send a collapse request $< \mathtt{corr}, \mathtt{all} >$. Using $\mathcal{A}$ it is possible to construct a malicious sender $\mathcal{A}_{\mathsf{Com}}$ that breaks the hiding of Π_{Com} with non-negligible probability.
Let $\mathcal{CH}$ be the challenger of the hiding game of Π_{Com}. $\mathcal{A}_{\mathsf{Com}}$ computes the following steps:

1. Compute v_k running $f_{\mathsf{ID}}(1^\lambda; r)$ where r is an uniformly chosen randomness and sends the messages $m_0 = 0^q$ and $m_1 = v_k$ to $\mathcal{CH}$.

2. Upon receiving $\tilde{\mathtt{com}}_k$ from $\mathcal{CH}$, $\mathcal{A}_{\mathsf{Com}}$ interacts with $\mathcal{A}$ compute the proof π. computing all the messages of $\mathsf{S}'_{\mathsf{NIZK}}$ following the steps described in $H_2^k(\lambda)$ (and in $H_2^{k-1}(\lambda)$) except for the k-th commitment for which she uses $\tilde{\mathtt{com}}_k$.

3. Emulation of the state $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$ of $\mathcal{P}_{\mathsf{NIZK}}$ after π is compute: acting as $\mathsf{S}'_{\mathsf{NIZK}}$ in $H_2^k(\lambda)$ (and in $H_2^{k-1}(\lambda)$) and secure erase the decommitment information $\{\mathsf{Dec}_j\}_{j=1}^{u \cdot t}$ (except for $\tilde{\mathsf{Dec}}_k$ that was never available to $\mathcal{A}_{\mathsf{Com}}$), set the state $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$ as described $H_2^k(\lambda)$ (and in $H_2^{k-1}(\lambda)$)) that is as described in Step 12 of Figure 2.1.

4. EXECUTION OF Γ^{V} :
   (a) Emulate the honest players acting as the honest player of Γ^{V} (as described in Section $H_2^k(\lambda)$ (and in $H_2^{k-1}(\lambda)$)).
   (b) After π is computed if $\mathcal{A}$ sends a collapse request $< \texttt{corr}, \texttt{all} >$, disclose the states of all the honest players $\mathsf{st}_{\mathsf{Pt}_1}, \ldots, \mathsf{st}_{\mathsf{Pt}_{|\mathcal{H}|}}$ and $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$.
5. When $\mathcal{A}$ stops, $\mathcal{A}_{\mathsf{Com}}$ outputs the outcome of $\mathcal{A}$.

$\mathcal{A}_{\mathsf{Com}}$ emulates the states of all the honest players $\mathsf{st}_{\mathsf{Pt}_1}, \ldots, \mathsf{st}_{\mathsf{Pt}_{|\mathcal{H}|}}$ in a perfect manner, since $\mathcal{A}_{\mathsf{Com}}$ just acts as the honest players in the execution of Γ^{V}. Moreover, $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$ after π is computed in Step 3 of the above procedure, corresponds to the state of an honest $\mathcal{P}_{\mathsf{NIZK}}$ in $H_2^k(\lambda)$ (and in $H_2^{k-1}(\lambda)$). The proof is concluded observing that if $\mathcal{CH}$ uses the message m_0 to compute $\tilde{\texttt{com}}_k$ then the reduction is distributed as H_2^{k-1} and as H_2^k otherwise. □

Claim 3. *Assume that Π_{pvWI} satisfies WI w.r.t. blockchain failure as in Definition 7, secure erasure, and the blockchain protocol Γ^{V} satisfies Assumption 1, then for every $x_{\mathsf{pvWI}}, w_0, w_1$ s.t. $(x_{\mathsf{pvWI}}, w_0) \in \mathcal{R}_{\mathsf{pvWI}}$ and $(x_{\mathsf{pvWI}}, w_1) \in \mathcal{R}_{\mathsf{pvWI}}$ it holds that $H_2(\lambda) \approx H_3(\lambda)$.*

Proof. Suppose by contradiction that the above claim does not hold, this implies that there exists an adversary $\mathcal{A}$ that is able to distinguish between $H_2(\lambda)$ and $H_3(\lambda)$. Note that $\mathcal{A}$ could wait until the protocol Π_{NIZK} ends and then can send a collapse request $< \texttt{corr}, \texttt{all} >$. Using $\mathcal{A}$ it is possible to construct a malicious verifier $\mathcal{A}_{\mathsf{pvWI}}$ that breaks the WI w.r.t. blockchain failure property of Π_{pvWI}. Let $\mathcal{CH}$ be the challenger of the WI w.r.t. blockchain failure game of Π_{pvWI}. $\mathcal{A}_{\mathsf{pvWI}}$ computes the following steps.

1. $\mathcal{A}_{\mathsf{pvWI}}$ acts as described in $H_2(\lambda)$ and $H_3(\lambda)$ until Step 6 of Figure 2.1. In particular, $\mathcal{A}_{\mathsf{pvWI}}$ computes the instance $x_{\texttt{com}}$ and the witness $w_{\texttt{com}}$ as explained, respectively, in Step 6 and in Steps 7, 14-21 of Figure 2.2.to $\mathcal{CH}$ on starting of the interaction with $\mathcal{CH}$.
2. $\mathcal{A}_{\mathsf{pvWI}}$ interacts as a proxy between $\mathcal{CH}$ and $\mathcal{A}$ for the messages $\pi^1_{\mathsf{pvWI}}, \ldots, \pi^{m-1}_{\mathsf{pvWI}}$, and interacting with the blockchain as a $\mathcal{P}_{\mathsf{pvWI}}$ would do upon request of $\mathcal{CH}$. two cases are possible. computation of i-th (for $i = 1, \ldots, m-1$) message of Π_{pvWI} $\mathcal{A}_{\mathsf{pvWI}}$ (that is acting as $\mathcal{P}_{\mathsf{NIZK}}$) will obtain w s.t. $(x, w) \in \mathcal{R}$ and sends $x_{\mathsf{pvWI}} = (x, x_{\texttt{com}}), w, w_{\texttt{com}}$ to $\mathcal{CH}$.
3. $\mathcal{A}$ chooses $(x, w) \in \mathcal{R}$ before the last message of Π_{NIZK} and therefore $\mathcal{A}_{\mathsf{pvWI}}$ (that is acting as $\mathcal{P}_{\mathsf{NIZK}}$) will obtain w s.t. $(x, w) \in \mathcal{R}$ and sends $x_{\mathsf{pvWI}} = (x, x_{\texttt{com}}), w, w_{\texttt{com}}$ to $\mathcal{CH}$ before that he receives π^m_{pvWI} from $\mathcal{CH}$. $\mathcal{A}_{\mathsf{pvWI}}$ completes the proof π as described in Step 12 of Figure 2.1.
   1. Emulation of the state $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$ of $\mathcal{P}_{\mathsf{NIZK}}$ after π is compute: $\mathcal{A}_{\mathsf{pvWI}}$ is acting as $\mathsf{S}'_{\mathsf{NIZK}}$ in $H_2(\lambda)$ (and in $H_3(\lambda)$) and secure erase the decommitment information $\{\mathsf{Dec}_j\}_{j=1}^{u \cdot t}$, set $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}} = \mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$, where $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$ is received from the challenger $\mathcal{CH}$.
4. EXECUTION OF Γ^{V} :
   1. $\mathcal{A}_{\mathsf{pvWI}}$ emulates the honest player acting as the honest player of Γ^{V} (as described in Section $H_3(\lambda)$ (and in $H_2(\lambda)$)).
   2. After π is computed if $\mathcal{A}$ sends a collapse request $< \texttt{corr}, \texttt{all} >$, $\mathcal{A}_{\mathsf{pvWI}}$ discloses the states of all the honest players $\mathsf{st}_{\mathsf{Pt}_1}, \ldots, \mathsf{st}_{\mathsf{Pt}_{|\mathcal{H}|}}$ and $\mathsf{st}_{\mathcal{P}_{\mathsf{NIZK}}}$.
5. When $\mathcal{A}$ stops, $\mathcal{A}_{\mathsf{pvWI}}$ outputs the outcome of $\mathcal{A}$.

We note that $\mathcal{A}_{\mathsf{pvWI}}$ simulates the states of all the honest players $\mathsf{st}_{\mathsf{Pt}_1}, \ldots, \mathsf{st}_{\mathsf{Pt}_{|\mathcal{H}|}}$ in a perfect way, this is because in the execution of Γ^{V}, $\mathcal{A}_{\mathsf{pvWI}}$ is behaving in the same way of the honest players of an execution of Γ^{V} (as described in $H_3(\lambda)$ (and in $H_2(\lambda)$)).

When $\mathcal{A}$ stops, $\mathcal{A}_{\mathsf{pvWI}}$ outputs the outcome of $\mathcal{A}$.

The proof is concluded observing that if $\mathcal{CH}$ uses the witness w to compute π_{pvWI} then the reduction is distributed as H_2, and as H_3 otherwise.

□

2.4 On Public Verifiability in [CGJ19]

A recent work [CGJ19] models the blockchain as a global ledger functionality $\mathcal{G}_{\texttt{ledger}}$ available to all the participants of a cryptographic protocol. [CGJ19] constructs concurrent self-composable secure computation protocol for general functionalities in such global ledger model. The protocols constructed in [CGJ19] are not publicly verifiable, and therefore do not satisfy the main feature that we study and achieve in this chapter. Indeed the authors of [CGJ19] already notice in their work that non-interactive zero knowledge for NP is impossible in their model. We remark that actually the impossibility extends also to publicly verifiable zero knowledge for languages that are not in BPP and we give now an high-level intuition. First of all, note that in the model of [CGJ19], since the blockchain is modeled as a global ledger, the simulator S of the zero-knowledge property has the same power of the adversary while accessing $\mathcal{G}_{\texttt{ledger}}$. Suppose now by contradiction that it is possible to construct a publicly verifiable zero-knowledge argument $\Pi = (\mathcal{P}, \mathcal{V})$ for the $\mathcal{NP}$-language $\mathcal{L}$ in the $\mathcal{G}_{\texttt{ledger}}$ model. This means that there exists a simulator S that having access to $\mathcal{G}_{\texttt{ledger}}$ on input any instance $x \in \mathcal{L}$ outputs an accepting proof π w.r.t. x that is (computationally) indistinguishable from a proof generated by a honest prover $\mathcal{P}$. Let us now consider a malicious polynomial-time prover $\mathcal{P}^*$ that in the $\mathcal{G}_{\texttt{ledger}}$-model wants to prove a false statement x^* to an honest verifier $\mathcal{V}$. We will show that $\mathcal{P}^*$ proves a false theorem with non-negligible probability, $\mathcal{P}^*$ works as follows. $\mathcal{P}^*$ internally runs S on input x^*. Moreover, each interaction that S wants to do with $\mathcal{G}_{\texttt{ledger}}$ is emulated by $\mathcal{P}^*$ and this is possible since S and $\mathcal{P}^*$ are accessing $\mathcal{G}_{\texttt{ledger}}$ in the same way. At the end of the execution, S outputs π^* w.r.t. x^*. $\mathcal{P}^*$ forwards π^* to $\mathcal{V}$.

Note that we are guaranteed by the zero-knowledge property that π^* is accepting and the view of an honest verifier that receives π^* from $\mathcal{P}^*$ is (computationally) indistinguishable from the view that $\mathcal{V}$ has when she receives a proof from an honest prover. Finally we note that public verifiability guarantees that π^* can be accepted by any verifier. The only caveat in the above reasoning can concern the fact that S might refuse to produce an accepting proof when $x \notin \mathcal{L}$. However this immediately shows that the language $\mathcal{L}$ is in BPP.

2.5 Publicly Verifiable WI of [SSV19]

In this section we will show that construction of delayed-input publicly verifiable witness indistinguishable proof system $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$ over any blockchain protocol $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GetHash})$ satisfying Assumption 1, presented in [SSV19] satisfies WI w.r.t. blockchain failure in the secure erasure model. The blockchain protocol Γ has chain consistency parameter $\eta(\lambda)$, pristine parameters t, n (for simplicity, we assume that n is an even integer).

The construction presented in [SSV19] makes use of the following tools. 1) A 3-round delayed-witness public-coin adaptive-input WI adaptive-input special-sound $\Pi_\Sigma = (\mathcal{P}_\Sigma, \mathcal{V}_\Sigma)$ for the relation $\mathcal{R}_{\mathsf{pvWI}}$. 2) An efficient procedure Ext (defined in [SSV19]) that takes as input t blocks, an auxiliary input aux and outputs τ strings $s_1, \ldots, s_\tau$ such that at least one string s_i is distributed statistically close to the uniform distribution over $\{0,1\}^\lambda$.

Π_{pvWI} is described in Figure 2.3.

Theorem 2. *Let* $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ *be a blockchain protocol that satisfies Assumption 1. Let* $\Pi_\Sigma = (\mathcal{P}_\Sigma, \mathcal{V}_\Sigma)$ *be a 3-round delayed-input public-coin adaptive-input WI adaptive-input special-sound* $\mathcal{R}_{\mathsf{pvWI}}$. *Assuming secure erasure* $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$ *is a delayed-input publicly verifiable WI w.r.t. blockchain failure proof system over* Γ^{V} *for* $\mathcal{NP}$.

Proof. The proofs of completeness follows from the proof of Theorem 2 in [SSV19].

WI w.r.t. blockchain failure satisfying Definition 7. In order to show that Π_{pvWI} enjoys WI w.r.t. blockchain failure satisfying Definition 7 we will consider the following 2 hybrid experiments.

Let $H_0(\lambda)$ be defined as the execution of Π_{pvWI}, where $\mathcal{P}_{\mathsf{pvWI}}$ uses the witness w_0. Let $H_1(\lambda)$ be defined as the execution of Π_{pvWI}, where $\mathcal{P}_{\mathsf{pvWI}}$ uses the witness w_1. Let $\mathcal{A}$ be the adversary as defined in Definition 7. The

DELAYED-INPUT PUBLICLY VERIFIABLE WI W.R.T. BLOCKCHAIN FAILURE PROOF SYSTEM $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$

Parameters : τ is a parameters of Ext, η is the chain consistency parameter of Γ^{V}, t, n are pristine parameters of Γ^{V}.

PROVER PROCEDURE: $\mathcal{P}_{\mathsf{pvWI}}$. Input: instance x, witness w s.t. $(x, w) \in \mathcal{R}_{\mathsf{pvWI}}$.

— **First message.**

1. Compute $\Sigma_i^1 \leftarrow \mathcal{P}_\Sigma(1^\lambda, x)$, for $i = 1, \ldots, \tau$.

— **Blockchain Interaction.**

2. Set $\mathsf{st} = \epsilon$. Post $\Sigma_1^1 || \ldots || \Sigma_\tau^1$ on the blockchain by running $\mathsf{Broadcast}(1^\lambda, \Sigma_1^1 || \ldots || \Sigma_\tau^1)$ and then monitor the blockchain by running $\mathsf{st} = \mathsf{UpdateState}(1^\lambda, \mathsf{st})$, $\mathbf{B} = \mathsf{GetRecords}(1^\lambda, \mathsf{st})$, until $\Sigma_1^1 || \ldots || \Sigma_\tau^1$ followed by t additional blocks $B_1, \ldots, B_t$ are posted on the blockchain $\mathbf{B}^{\lceil \eta}$.

— **Second message.**

3. Let $B_{p_1}, \ldots, B_{p_\tau}$ be the pristine blocks in the sequence $B_1, \ldots, B_t$. Extract challenges by executing $\mathsf{Ext}(B_{p_1}, \ldots, B_{p_n}, \mathsf{aux})$ and obtain $r_1, \ldots, r_\tau$. Set $\Sigma_i^2 = r_i$ for $i = 1, \ldots, \tau$.
4. Compute $\Sigma_i^3 \leftarrow \mathcal{P}_\Sigma(\Sigma_i^2, x_\Sigma, w)$, for $i = 1, \ldots, \tau$.

— **Blockchain Interaction.**

5. Post $\Sigma_1^3 || \ldots || \Sigma_\tau^3$ on the blockchain by running $\mathsf{Broadcast}(1^\lambda, \Sigma_1^3 || \ldots || \Sigma_\tau^3)$ and then monitor the blockchain by running $\mathsf{st} = \mathsf{UpdateState}(1^\lambda, \mathsf{st})$, $\mathbf{B} = \mathsf{GetRecords}(1^\lambda, \mathsf{st})$, until $\Sigma_1^3 || \ldots || \Sigma_\tau^3$ is posted on the blockchain $\mathbf{B}^{\lceil \eta}$.
6. Set $\pi = (x, \{\Sigma_i^1, \Sigma_i^2, \Sigma_i^3\}_{i=1}^\tau)$ and output π erasing all the coins needed to compute π setting $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$ as empty.

VERIFIER PROCEDURE: $\mathcal{V}_{\mathsf{pvWI}}$. Input: x, $\pi = (x, \{\Sigma_i^1, \Sigma_i^2, \Sigma_i^3\}_{i=1}^\tau)$, and a blockchain $\tilde{\mathbf{B}}$ works as follows.

— **Check Blockchain.** If the messages $\{\Sigma_i^1\}_{i=1}^\tau$ are not posted on the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ then $\mathcal{V}_{\mathsf{pvWI}}$ outputs 0. Otherwise, let B^* be the block of the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ where the messages $\{\Sigma_i^1\}_{i=1}^\tau$ are posted. Let $B_1, \ldots, B_t$ be t consecutive blocks of the blockchain $\tilde{\mathbf{B}}^{\lceil \eta}$ after B^* ,and let $B_{p_1}, \ldots, B_{p_n}$ be the pristine blocks. $\mathcal{V}_{\mathsf{pvWI}}$ computes $\{\Sigma_i^2\}_{i=1}^\tau = \mathsf{Ext}(B_{p_1}, \ldots, B_{p_n}, \mathsf{aux})$.

— **Check Proof.** Accept if all the following conditions are satisfied.
- The messages $\{\Sigma_i^3\}_{i=1}^\tau$ are posted at least t blocks after B^*;
- $\mathcal{V}_\Sigma(x, \Sigma_i^1, \Sigma_i^2, \Sigma_i^3) = 1$ for $i = 1, \ldots, \tau$;

EXECUTION OF Γ^{V} BY HONEST PLAYER Pt_j:

Pt_j acts as described in Section 2.2.2.

If $\mathcal{A}$ sends a collapse request $< \mathtt{corr}, \mathtt{all} >$, $\mathcal{A}$ obtains $\mathsf{st}_{\mathsf{Pt}_i}$ from honest player Pt_i, for all $i = 1, \ldots, |\mathcal{H}|$, moreover $\mathcal{A}$ obtains the state $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$ of $\mathcal{P}_{\mathsf{pvWI}}$.

Figure 2.3: Delayed-input publicly verifiable WI w.r.t. blockchain failure proof system Π_{pvWI}.

output of each experiment is the pair $(\pi, \mathsf{view}_{\mathcal{A}})$, where π is the transcript of Π_{pvWI} computed in the experiment and $\mathsf{view}_{\mathcal{A}}$ is the view of $\mathcal{A}$ in the experiment.

Claim 4. *Let $\Pi_\Sigma = (\mathcal{P}_\Sigma, \mathcal{V}_\Sigma)$ be a 3-round delayed-witness public-coin adaptive-input WI adaptive-input special-sound $\mathcal{R}_{\mathsf{pvWI}}$. Assuming secure erasure, and Assumption 1 holds for Γ^{V} then for every $x_{\mathsf{pvWI}}, w_0, w_1$ s.t. $(x_{\mathsf{pvWI}}, w_0) \in \mathcal{R}_{\mathsf{pvWI}}$ and $(x_{\mathsf{pvWI}}, w_1) \in \mathcal{R}_{\mathsf{pvWI}}$ it holds that $H_0(\lambda) \approx H_1(\lambda)$.*

Proof. Suppose by contradiction that the above claim does not hold, this implies that there exists an adversary $\mathcal{A}$ that is able to distinguish w.r.t. between $H_0(\lambda)$ and $H_1(\lambda)$ Note that $\mathcal{A}$ has the additional power to wait until the protocol Π_{pvWI} ends and then sends a collapse request $< \mathtt{corr}, \mathtt{all} >$.

Let $\mathcal{CH}$ be the challenger of adaptive-input WI game of Π_Σ. $\mathcal{A}_\Sigma$ will interact as a proxy between $\mathcal{CH}$ and $\mathcal{A}$ for the messages $\{\Sigma_i^1, \Sigma_i^3\}_{i=1}^\tau$ and she will compute all other messages following $\mathcal{P}_{\mathsf{pvWI}}$ of H_0 (of H_1).

In more details, $\mathcal{A}_\Sigma$ acts as follows.

1. $\mathcal{A}_\Sigma$ receives $\{\tilde{\Sigma}_i^1\}_{i=1}^\tau$ from $\mathcal{CH}$ and sets $\Sigma_i^1 = \tilde{\Sigma}_i^1$ for $i = \{1, \ldots, \tau\}$.
2. $\mathcal{A}_\Sigma$ computes the other steps of Π_{pvWI}, until Step 3, she acts as $\mathcal{P}_{\mathsf{pvWI}}$ of H_0 (of H_1). In particular in Step 3 $\mathcal{A}_\Sigma$ computes $\{\Sigma_i^2\}_{i=1}^\tau$ as $\mathcal{P}_{\mathsf{pvWI}}$ of H_0 (of H_1) does.
3. $\mathcal{A}_\Sigma$ sends $\{\Sigma_i^2\}_{i=1}^\tau$ to $\mathcal{CH}$ along with x, w_0, w_1 obtained from $\mathcal{A}$.
4. $\mathcal{A}_\Sigma$ receives $\{\tilde{\Sigma}_i^3\}_{i=1}^\tau$ from $\mathcal{CH}$ and sets $\Sigma_i^3 = \tilde{\Sigma}_i^3$ for $i = \{1, \ldots, \tau\}$, $\mathcal{A}_\Sigma$ completes the computations of π precisely as $\mathcal{P}_{\mathsf{pvWI}}$ does in both H_0 and H_1.
5. Emulate state $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$ of $\mathcal{P}_{\mathsf{pvWI}}$: $\mathcal{A}_\Sigma$ acting as $\mathcal{P}_{\mathsf{pvWI}}$ erases all the coins needed to compute π as soon as the proof is computed, setting $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$ as an empty state,
6. If $\mathcal{A}$ sends a corruption request $< \mathtt{corr}, \mathtt{all} >$ $\mathcal{A}_\Sigma$ discloses $\mathsf{st}_{\mathcal{P}_{\mathsf{pvWI}}}$. $\mathcal{A}_\Sigma$ emulates the state of $\mathcal{P}_{\mathsf{pvWI}}$ in a perfect way, since at that point (after the computation of π) the state of honest player $\mathcal{P}_{\mathsf{pvWI}}$ is empty since she already erased the coins of π.

   At the end of the execution $\mathcal{A}_\Sigma$ outputs what $\mathcal{A}$ outputs.

The proof is concluded observing that if $\mathcal{CH}$ uses the witness w_0 to compute $\{\tilde{\Sigma}_i^3\}_{i=1}^\tau$ then the output of this execution is distributed as H_0. Instead if $\mathcal{CH}$ uses the witness w_1 to compute $\{\tilde{\Sigma}_i^3\}_{i=1}^\tau$ then the output of this execution is distributed as H_1. □

Soundness, Definition 5. We note that Π_{pvWI} satisfies the Definition 5, it follows a sketched proof.

Let $\mathcal{P}^\star$ be a successful adversary. Recall that $\mathcal{P}^\star$ is successful if it produces with non-negligible probability an accepting π of Π_{pvWI} w.r.t. $x \notin \mathcal{L}_{\mathsf{pvWI}}$, where x is adaptively chosen by $\mathcal{P}^\star$ before the last message of π.

We will now argue that the probability with which $\mathcal{P}^\star$ completes the execution of Π_Σ w.r.t. x_{pvWI} s.t. $x_{\mathsf{pvWI}} \notin \mathcal{L}_\Sigma$ is negligible in λ. First, note that at least one of the outputs of $\mathsf{Ext}(B_{p_n}, \ldots, B_{p_n}, \mathsf{aux})$ is distribute statistically close to a distribution over $\{0,1\}^\lambda$. In particular, let Σ_i^{*2} be this output, from the adaptive-input special soundness of Π_Σ it follows that $\mathcal{P}^\star$ computes Σ_i^3 s.t. $\mathcal{V}_\Sigma(x_{\mathsf{pvWI}}, \Sigma_i^1, \Sigma_i^{*2}, \Sigma_i^3) = 1$ with probability less or equal to $2^{-\lambda}$.

Summing up, since x_{pvWI} is s.t. $x_{\mathsf{pvWI}} \notin \mathcal{L}_\Sigma$ with non-negligible probability and $\mathcal{P}^*$ has only negligible probability to compute τ accepting transcripts of Π_Σ w.r.t a false instance x_{pvWI}, we can conclude that $\mathcal{P}^*$ is successful only with negligible probability.

□

Note that is ti possible to instantiate $\Pi_\Sigma = (\mathcal{P}_\Sigma, \mathcal{V}_\Sigma)$ using [LS90] that is adaptive-input special-sound in the variant of [COSV17].

Corollary 2. *Let $\Gamma^{\mathsf{V}} = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast}, \mathsf{GenBlock})$ be a blockchain protocol that satisfies Assumption 1. Assuming secure erasure $\Pi_{\mathsf{pvWI}} = (\mathcal{P}_{\mathsf{pvWI}}, \mathcal{V}_{\mathsf{pvWI}})$ is a delayed-input publicly verifiable WI w.r.t. blockchain failure proof system over Γ^{V} for $\mathcal{NP}$.*

Chapter 3

Quick Computations on Blockchains

We consider a general transform to design smart contracts that retain security in the presence of forks. As security notion for modeling executions of smart contracts, we focus on secure multi-party computation (MPC). In particular we consider on-chain MPC executions with the aid of smart contracts. The classical double-spending problem tells us that messages of the MPC protocol should be confirmed on-chain before playing the next ones, thus slowing down the entire execution. We show how to design smart contracts on forking blockchains reducing the number of confirmations, still maintaining security and fairness.

We design a compiler that takes any "digital and universally composable" MPC protocol (with or without honest majority) and transforms it into another one (for the same task and same setup) where all messages are played on-chain without delays and still security is maintained. The special requirements on the starting protocol mean that messages consists only of bits (e.g., no hardware token is sent) and security holds also in the presence of other protocols. Then we show that our compiler satisfies fairness with penalties as long as honest players only wait once. By reducing the number of confirmations, we construct protocols that are significantly faster than previous constructions.

The full version of the results presented in this chapter can be found here [BFVV19].

3.1 Running MPC on Forking Blockchains

Given the definitions of the blockchain model reported in Section 2.2.1, we formalize different ways how to run an MPC protocol with the aid of a blockchain.

In Section 3.1.1 we specify what it means to run an MPC protocol on the blockchain both in the presence of quick and non-quick players. The security definition appears in Section 3.1.2.

3.1.1 Blockchain-Aided MPC

Next, we define what it means to run an n-party protocol π for securely computing some function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ over a blockchain protocol Γ.

Intuitively, running π on Γ simply means that the players write the protocol's messages on the blockchain instead of using point-to-point connections. However, since the blockchain may fork, the protocol's participants have to choose how to manage possibly unconfirmed blocks that are part of the current chain. Looking ahead, this choice will have impact both on the efficiency and on the security of the protocol execution. In particular, we distinguish between *quick* and *non-quick* players as formalized below.

Non-quick execution. Roughly speaking, a player is said to be *non-quick* if it always decides its next message by looking at the transcript of the protocol that is obtained by pruning the last k blocks of the blockchain, where k is the parameter for the consistency property of the underlying blockchain.

Definition 9 (Non-quick player)**.** Let $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol with k-consistency. A player P_i is said to be *non-quick* if it behaves as follows:

- Initialize $\tau_i^{(0)} := \varepsilon$, $\mathsf{st}_i := \varepsilon$ and $r_i := 0$.
- Run the following loop:
  - Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and retrieve $\mathcal{B}_i \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_i)$ until the partial transcript $\tau^{(r_i)}$ is contained in $\mathcal{B}_i^{\lceil k}$.
  - If the protocol is over (i.e., the transcript $\tau^{(r_i)}$ is sufficient for determining the output), output the value y_i as a function of $\tau_i^{(r_i)}$ and terminate.
  - Else, compute the next protocol message $m_i^{(r_i+1)}$, invoke $\mathsf{Broadcast}(m_i^{(r_i+1)})$, and set $r_i := r_i + 1$.

Quick execution. On the other hand, a player is *quick* if it decides and broadcasts its next message by looking at the latest version of the blockchain (i.e., without pruning blocks). Since the consistency property does not hold for the last k blocks, quick players may retrieve different protocol's transcripts as the protocol proceeds. In particular, it may happen that at a given time step party P_i reads from the blockchain a partial transcript $\tau^{(\tilde{r})}$, whereas at a later time step the same player reads $\tau^{(\tilde{r}')}$ for some $\tilde{r}' < \tilde{r}$. This is due to the fact that some of the messages contained in $\tau^{(\tilde{r})}$ may end up in unconfirmed blocks, and thus be discarded.

Definition 10 (Quick execution)**.** Let $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol with k-consistency. A player P_i is said to be *quick* if it behaves as follows:

- Initialize $\tau_i^{(0)} := \varepsilon$ and $\mathsf{st}_i := \varepsilon$.
- Run the following loop:
  - Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_i \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_i)$.
  - Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_i$.
  - If the protocol is over (i.e., the transcript $\tau^{(\tilde{r})}$ is sufficient for determining the output), output the value y_i as a function of $\tau^{(\tilde{r})}$ and terminate.
  - Else, compute the next protocol message $m_i^{(\tilde{r}+1)}$ and invoke $\mathsf{Broadcast}(m_i^{(\tilde{r}+1)})$.

More generally, we call φ-quick a player that is non-quick until a partial transcript $\tau^{(\varphi)}$ is at least k blocks deep in the blockchain, and afterwards it starts being quick. We sometimes call φ the *finality parameter*. Note that a 0-quick player is identical to a quick player, whereas an ∞-quick player is identical to a non-quick player. We call (χ, φ)-quick a player that is quick for the first χ rounds, and then behaves like a φ-quick player.

3.1.2 Security in the Presence of Quick Players

We can now define security of MPC protocols running on the blockchain. As in the standard setting, the definition compares a protocol execution in the real world with one in the ideal setting where a trusted party is made available. The main difference with the standard definition is that the attacker A is given black-box access to the algorithms in Γ, which it can use arbitrarily. The simulator is not allowed to control the blockchain (i.e. it must simulate the view of the adversary while invoking the algorithms in Γ on behalf of the honest players).

The real model: This is the execution of π on Γ, where the honest players are φ-quick. As usual, the adversary A is coordinated by a non-uniform distinguisher D. At the outset, D chooses the inputs $(1^\lambda, x_i)$ for each player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i\in\mathcal{I}}$ and z to A, where $\mathcal{I} \subseteq [n]$ represents the set of corrupted players and z is some auxiliary input. The parties then start running π on Γ, with the honest players P_i being φ-quick and behaving as prescribed in π (using input x_i), and with malicious parties behaving arbitrarily (directed

by A). At some point, A gives to D an arbitrary function of its view; note that the latter includes the view generated via $\mathbf{Exec}_{\Gamma,\mathsf{A},\mathsf{D}}(\lambda)$ in the blockchain protocol. Finally, D receives the outputs of the honest parties and must output a bit. We denote by $REAL^{\Gamma,\varphi}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)$ the random variable corresponding to D's guess.

The ideal model: This is identical to the ideal model for standard MPC (Section 3.1.4), with the only difference that the simulator S is also responsible for simulating the attacker's view corresponding to the interaction of the honest players with the blockchain. The latter is achieved using the algorithms of the underlying blockchain protocol Γ. We denote by $IDEAL^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ and $IDEAL^{\Gamma}_{f_\perp,\mathsf{S},\mathsf{D}}(\lambda)$ the random variable corresponding to D's guess in the ideal world, where the latter is for the case of security with aborts.

Definition 11 (Secure MPC in the presence of quick players)**.** Let π be an n-party protocol run over a blockchain protocol Γ. We say that π t-securely computes f in the presence of φ-quick players and malicious adversaries if for every PPT adversary A there exists a PPT simulator S such that for every non-uniform PPT distinguisher D corrupting at most t parties the following holds:

$$\left\{REAL^{\Gamma,\varphi}_{\pi,\mathsf{A},\mathsf{D}}(\lambda)\right\}_{\lambda\in\mathbb{N}} \approx_c \left\{IDEAL^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)\right\}_{\lambda\in\mathbb{N}}.$$

When replacing $IDEAL^{\Gamma}_{f,\mathsf{S},\mathsf{D}}(\lambda)$ with $IDEAL^{\Gamma}_{f_\perp,\mathsf{S},\mathsf{D}}(\lambda)$ we say that π t-securely computes f *with aborts* in the presence of φ-quick players and malicious adversaries.

Remark 1 (On $\varphi = \infty$). One may think that every protocol π that t-securely computes f (with or without aborts) in the presence of malicious adversaries, must t-securely compute f (with or without aborts) in the presence of ∞-quick (i.e., non-quick) players and malicious adversaries.

Remark 2 (On $\varphi = 0$). Note that when the players are fully quick (i.e., $\varphi = 0$), the adversary's view in the real world may include multiple executions of the original protocol π (upon the same inputs chosen by the distinguisher). This view may not be possible to simulate in the ideal world, where the simulator can invoke the ideal functionality f only once.

For this reason, whenever $\varphi = 0$, we implicitly assume that the simulator is allowed to query the ideal functionality f multiple times. Note that this yields a meaningful security guarantee only for certain functionalities f, similarly to the setting of resettably secure computation [GS09].

Remark 3 (On the power of the adversary). We stress that we assume that the adversary of the MPC protocol has no impact on the execution of the consensus protocol of the underlying blockchain. Note that if we would instead assume that the adversary of the MPC protocol also creates new branches and/or contributes in deciding which branch of a fork is eventually confirmed on the blockchain then he can have an unfair advantage. Indeed the adversary can start more branches when he does not like the output computed in a branch, and/or can decide which output among the various outputs appearing in different branches should be confirmed on the blockchain. Obviously the above unfair advantages are unavoidable and our protocol is still secure by introducing the unavoidable real-world attack into the ideal world, similarly to the classical fairness issue resolved through aborts in the ideal world.

Remark 4 (On public verifiability). We notice that any on-chain MPC protocol with quick players admits the case where a honest player complete her execution computing an output that does not necessarily correspond to the transcript that others later on will see on the blockchain. In other words, the local output computed by players could not match the publicly verifiable execution that remains visible on the blockchain. The reason why public verifiability could fail is that an execution of the protocol could be entirely contained in a branch of a fork that will not become permanent in the blockchain. The above issue is intrinsic in all protocols played on-chain in the presence of forks and quick players. An obvious solution for a honest player consists of waiting that the last message of the protocol is confirmed on the blockchain and only after that the computation ends returning the computed output.

3.1.3 Preliminaries

In the next subsection we need to use the following cryptographic tools.

Public-key encryption. A public-key encryption (PKE) scheme is a tuple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ specified as follows. (i) The randomized algorithm Gen takes as input the security parameter, and outputs a pair of keys (pk, sk); (ii) The randomized algorithm Enc takes as input a public key pk and a message $m \in \mathcal{M}$, and outputs a ciphertext c; (iii) The deterministic algorithm Dec takes as input a secret key sk and a ciphertext c, and outputs a value in $\mathcal{M} \cup \{\perp\}$ (where $\perp$ denotes decryption error). Correctness says that for every key $\lambda \in \mathbb{N}$, every (pk, sk) in the support of $\mathsf{Gen}(1^\lambda)$, and every message $m \in \mathcal{M}$, it holds that $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m$ with probability one over the randomness of Enc.

Definition 12 (Semantic security)**.** We say that $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ satisfies semantic security if for all PPT attackers $\mathsf{A} := (\mathsf{A}_0, \mathsf{A}_1)$ there exists a negligible function $\nu(\cdot)$ such that:

$$\left| \mathbb{P}\left[b' = b : \begin{array}{c} (pk, sk) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda); (m_0, m_1, z) \leftarrow\!\!\$\ \mathsf{A}_0(pk) \\ b \leftarrow\!\!\$\ \{0,1\}; c \leftarrow\!\!\$\ \mathsf{Enc}(pk, m_b); b' \leftarrow\!\!\$\ \mathsf{A}_1(z, c) \end{array} \right] - \frac{1}{2} \right| \leq \nu(\lambda).$$

Signature schemes. A signature scheme is a tuple of polynomial-time algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ specified as follows. (i) The randomized algorithm Gen takes as input the security parameter and outputs a secret key sk together with a public verification key pk; (ii) The deterministic algorithm Sign takes as input the secret key sk and a message $x \in \{0,1\}^*$ and outputs a signature y; (iii) The randomized algorithm Verify takes as an input the verification key pk, a message/signature pair (x, y) and outputs a decision bit.

Correctness says that for all $\lambda \in \mathbb{N}$, for all $(pk, sk) \in \mathsf{Gen}(1^\lambda)$, and for all $x \in \{0,1\}^*$ it holds that $\mathsf{Verify}(pk, x, \mathsf{Sign}(sk, x)) = 1$ (with probability one over the coin tosses of Verify).

Secret Sharing Schemes. An n-party secret sharing scheme $(\mathsf{Share}, \mathsf{Recon})$ is a pair of poly-time algorithms specified as follows. (i) The randomized algorithm Share takes as input a message $m \in \mathcal{M}$ and outputs n shares $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$; (ii) The deterministic algorithm Recon takes as input a subset of the shares, say $\sigma_{\mathcal{I}}$ with $\mathcal{I} \subseteq [n]$, and outputs a value in $\mathcal{M} \cup \{\perp\}$.

Definition 13 (Threshold secret sharing)**.** Let $n \in \mathbb{N}$. For any $t \leq n$, we say that $(\mathsf{Share}, \mathsf{Recon})$ is an (t, n)-secret sharing scheme if it satisfies the following properties.

- **Correctness**: For any message $m \in \mathcal{M}$, and for any $\mathcal{I} \subseteq [n]$ such that $|\mathcal{I}| \geq t$, we have that $\mathsf{Recon}(\mathsf{Share}(m)_{\mathcal{I}}) = m$ with probability one over the randomness of Share.
- **Privacy**: For any pair of messages $m_0, m_1 \in \mathcal{M}$, and for any $\mathcal{U} \subset [n]$ such that $|\mathcal{U}| < t$, we have that

$$\{\mathsf{Share}(1^\lambda, m_0)_{\mathcal{U}}\}_{\lambda \in \mathbb{N}} \approx_c \{\mathsf{Share}(1^\lambda, m_1)_{\mathcal{U}}\}_{\lambda \in \mathbb{N}}.$$

3.1.4 Multi-Party Computation

We recall standard notion of UC-security for multi-party computation (MPC). Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be a function, and consider n players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ executing a protocol π for computing f. Our default network model consists of the players interacting in synchronous rounds via private and authenticated point-to-point channels.

Intuitively, the security of π is formalized by comparing its execution in the real world (where an attacker may corrupt a subset of the players) with the ideal execution in which a trusted party computes the function f on behalf of the players.

The real model. In the real world, the protocol π is run in the presence of an adversary A coordinated by a non-uniform environment $\mathsf{Z} = \{\mathsf{Z}_\lambda\}_{\lambda\in\mathbb{N}}$. At the outset, Z chooses the inputs $(1^\lambda, x_i)$ for each player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i\in\mathcal{I}}$ and z to A, where $\mathcal{I} \subseteq [n]$ represents the set of corrupted players and z is some auxiliary input. For simplicity, we only consider static corruptions (i.e., the environment decides who is corrupt at the beginning of the protocol). The parties then start running π, with the honest players P_i behaving as prescribed in the protocol (using input x_i), and with malicious parties behaving arbitrarily (directed by A). The attacker may delay sending the messages of the corrupted parties in any given round until after the honest parties send their messages in that round; thus, for every r, the round-r messages of the corrupted parties may depend on the round-r messages of the honest parties.

At some point, A gives to Z an arbitrary function of its view, and Z additionally receives the outputs of the honest parties and must output a bit. We denote by $REAL_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's guess.

The ideal model. In the ideal world, a trusted third party evaluates the function f on behalf of a set of dummy players $(\mathsf{P}_i)_{i\in[n]}$. As in the real setting, Z chooses the inputs $(1^\lambda, x_i)$ for each honest player P_i, and gives $\mathcal{I}$, $\{x_i\}_{i\in\mathcal{I}}$ and z to the ideal adversary S, corrupting the dummy parties $(\mathsf{P}_i)_{i\in\mathcal{I}}$. Hence, honest parties send their input $x'_i = x_i$ to the trusted party, whereas the parties controlled by S might send an arbitrary input x'_i. The trusted party computes $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$, and sends y_i to P_i. Finally, S gives to Z an arbitrary function of its view, and Z additionally receives the outputs of the honest parties and must output a bit. We denote by $IDEAL_{f,\mathsf{S},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's guess.

The above specification of the ideal model automatically implies *fairness* (i.e., corrupted parties get the output if and only if honest parties do as well). Unfortunately, as shown by Cleve [Cle86], such a strong guarantee is impossible to achieve for some functionalities without assuming honest majority. For this reason, we also consider a weaker flavor of the ideal model yielding a middle-ground notion known as security with aborts, which is possible to achieve even in the presence of honest minority. Let $\mathcal{H} := [n] \setminus \mathcal{I}$. The only difference with the above specification is that the trusted party at first forwards only the outputs $\{y_i\}_{i\in\mathcal{I}}$ to the ideal adversary S. Hence, S might send either a message $(\texttt{continue}, \mathcal{H}')$ or $\texttt{abort}$ to the trusted party. In the former case, all the honest parties in $\mathcal{H}'$ are given their output y_i whereas the honest parties in $\mathcal{H} \setminus \mathcal{H}'$ receive an abort symbol $\perp$. In the latter case, all honest parties receive $\perp$. We denote by $IDEAL_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ the random variable corresponding to Z's final guess.

The definition. We are now ready to define security.

Definition 14 (UC-Secure MPC)**.** Let π be an n-party protocol for computing a function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$. We say that π t-securely UC-realizes f in the presence of malicious adversaries if for every PPT adversary A there exists a PPT simulator S such that for every non-uniform PPT environment Z corrupting at most t parties the following holds:

$$\{REAL_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{IDEAL_{f,\mathsf{S},\mathsf{Z}}(\lambda)\}_{\lambda\in\mathbb{N}}.$$

When replacing $IDEAL_{f,\mathsf{S},\mathsf{Z}}(\lambda)$ with $IDEAL_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ we say that π t-securely computes f *with aborts* in the presence of malicious adversaries.

3.2 Compiler Description

In this section, we propose and analyze a simple transformation that allows to run any MPC protocol safely on the blockchain, even when the players are quick. The description of our compiler appears in Section 3.2.1, while in Section 3.2.2 we analyze its security. Finally, in Section 3.2.3, we discuss how to extend our generic transformation in order to achieve fairness with penalties, as long as the players start being quick after the confirmation of the first round.

3.2.1 Compiler Description

Intuitively our transformation proceeds as follows. Our starting point is any MPC protocol π UC-securely computing an n-party functionality $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ in the presence of malicious players (with aborts). Hence, the honest players fix the random tape for running π and simply execute protocol π by broadcasting their messages on the blockchain. Furthermore, each honest player P_i keeps track of the longest protocol transcript α_i generated so far and, in the presence of a fork, aborts the execution in case the view on a given branch is not consistent with α_i[1]. This intuitively ensures that the underlying protocol π is run only once, even in the presence of forks.

Since the initial protocol π may require private channels between the players, we need to augment the above transformation in such a way that subsets of honest parties can exchange messages in a confidential and authenticated manner. Let $m_{i,j}^{(r)}$ be the message that P_i sends to P_j at the generic round r. The latter is achieved by having P_i encrypting $m_{i,j}^{(r)}$ using the public encryption key ek_j of P_j, and then signing the resulting ciphertext $c_{i,j}^{(r)}$ with its own private signing key sk_i, which is the standard way of building a secure channel. We refer the reader to Fig. 3.1 for a formal description.

Generic Compiler π^*

Let π be an n-party ρ-round protocol, and $\Gamma = (\mathsf{UpdateState}, \mathsf{GetRecords}, \mathsf{Broadcast})$ be a blockchain protocol. Further, let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a PKE scheme and $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme, both with domain $\{0,1\}^*$ (see Section 3.1.3 for the formal definitions). The protocol π^* proceeds as follows:

- For $i \in [n]$, each player P_i initializes $\mathsf{st}_i := \varepsilon$, samples $(ek_i, dk_i) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$, $(\mathsf{vk}_i, sk_i) \leftarrow\!\!\$\ \mathsf{Gen}'(1^\lambda)$, and $\omega_i \leftarrow\!\!\$\ \{0,1\}^*$, and invokes $\mathsf{Broadcast}(ek_i||\mathsf{vk}_i||i)$.
- For $i \in [n]$, each player P_i keeps running $\mathsf{st}_i \leftarrow\!\!\$\ \mathsf{UpdateState}(1^\lambda)$ and $\mathcal{B}_i \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_i)$ until all the messages $(ek_j, \mathsf{vk}_j)_{j\in[n]} \in \mathcal{B}_i$.
- For $i \in [n]$, each player P_i sets $\tau^{(0)} := (ek_j, \mathsf{vk}_j)_{j\in[n]}$ and $\alpha_i := \tau^{(0)}$, and then runs the following loop:
  1. Update the state st_i by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_i \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_i)$.
  2. Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_i$. Then:
     - If the ciphertexts in $\tau^{(\tilde{r})}$ are not consistent with those in α_i, output $\perp$ and terminate.
     - Else if $\tilde{r} = \rho$, output the value y_i as a function of $\tau^{(\rho)}$ and terminate.
     - Else, go to the next step and if α_i is a prefix of $\tau^{(\tilde{r})}$ let $\alpha_i := \tau^{(\tilde{r})}$.
  3. For each $j \in [n]$, with $j \neq i$, and for each $r \leq \tilde{r}$, decrypt the ciphertexts $c_{j,i}^{(r)}$ and use the corresponding values $m_{j,i}^{(r)}$ to compute the messages $m_{i,j}^{(\tilde{r}+1)}$ to be sent at round $\tilde{r}+1$ (using the corresponding portion of the random tape ω_i).
  4. Finally, let $c_{i,j}^{(\tilde{r}+1)} \leftarrow\!\!\$\ \mathsf{Enc}(ek_j, m_{i,j}^{(\tilde{r}+1)})$ (using again random coins coming from ω_i) and $\sigma_{i,j}^{(\tilde{r}+1)} = \mathsf{Sign}(sk_i, c_{i,j}^{(\tilde{r}+1)})$, and invoke $\mathsf{Broadcast}((c_{i,j}^{(\tilde{r}+1)}||\sigma_{i,j}^{(\tilde{r}+1)})_{j\in[n]\setminus\{i\}})$.

Figure 3.1: Generic compiler for obtaining blockchain-aided MPC with quick players.

Note that wlog. we assume that in the underlying protocol π in each round every P_i sends a single message to each $\mathsf{P}_{j\neq i}$ over a private and authenticated channel. π^* takes the messages of the protocol π and send them on-chain. Moreover, P_i picks a sufficiently long random tape ω_i that is then used to run the compiled version π^* of π over Γ. Observe that ω_i includes both the randomness required to compute the messages in π and the random coins used to encrypt them. In π^*, in the presence of forks, an honest P_i that does not abort broadcasts on the blockchain exactly the same ciphertexts on multiple branches since the randomness used for encryption is the same on all branches where an honest party sends the message of π.

[1] We say that two views α_i and α_j are consistent if and only if α_j contains all the MPC protocol messages in α_i or α_i contains all the MPC protocol messages in α_j.

3.2.2 Security Analysis

The theorem below establishes the security of our generic compiler.

Theorem 3. *Let* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a semantically secure PKE scheme, and* $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$ *be a (deterministic) unforgeable signature scheme. Furthermore, let* π *be an* n*-party* ρ*-round protocol that* t*-securely UC-realizes a functionality* f *with aborts in the presence of malicious adversaries. Then, the protocol* π^* *of Fig. 3.1* t*-securely computes* f *with aborts in the presence of quick players and malicious adversaries.*

UC security is needed due to the fact that the attacker in the real world may interact with the blockchain by posting messages and reading its state. As shown in [CGJ19], such blockchain-active adversaries render standard simulation techniques (e.g., black-box rewinding) moot. Note also that Remark 2 does not hold for our protocol. If the adversary tries to furnish two different inputs in two different branches it can be spotted by some honest player, leading to an abort. Therefore only one possible input can be given to the functionality.

We need to show that for every PPT adversary A^*, there exists a PPT simulator S^* such that no non-uniform PPT distinguisher D^* can tell apart the experiments $REAL^{\Gamma,0}_{\pi,\mathsf{A}^*,\mathsf{D}^*}(\lambda)$ and $IDEAL^{\Gamma}_{f_\perp,\mathsf{S}^*,\mathsf{D}^*}(\lambda)$. In particular, the simulator S^* needs to simulate the interaction of the honest players with the blockchain protocol Γ as it happens in the real experiment. Intuitively, S^* relies on the simulator S guaranteed by the underlying protocol π as follows. At the beginning, S^* samples the public/secret keys for encryption/signatures for the honest players. Then, S^* runs A^* reading its messages from the emulated execution of the blockchain protocol Γ, and simulates its view as follows: (i) The round-r messages $m^{(r)}_{j,i}$ sent by the honest players P_j to the malicious players P_i are obtained from the simulator S; (ii) The round-r messages $m^{(r)}_{j,j'}$ that are exchanged by the honest players $\mathsf{P}_j, \mathsf{P}_{j'}$ are replaced with the all-zero string. Of course, S^* does additional bookkeeping in order to simulate a real execution of the protocol using the blockchain; in particular, S^* needs to check that the attacker plays consistently on different branches of a fork, and simulate an abort whenever the latter does not happen. Moreover, when S extracts the inputs for the malicious parties, the simulator S^* forwards the same inputs to the trusted party, obtains the outputs for the malicious parties, and sends it to S. Finally, S^* completes the simulation consistently with the choice of S of aborting or not.

Very roughly, the security of the PKE scheme and of the signature scheme imply that the view of the attacker is identical to that in a real execution of protocol π, so that security of π^* follows by that of π.

Proof. We begin by describing the simulator S^*. Let S be the PPT simulator guaranteed by the malicious security of π. Upon input the set of corrupted parties $\mathcal{I}$, inputs $(x_i)_{i\in\mathcal{I}}$, and auxiliary input z, the simulator S^* proceeds as follows:

1. Initialize S upon input $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$, with uniformly chosen random tape $\omega_{\mathsf{sim}} \leftarrow\!\!\$\ \{0,1\}^*$.
2. For each $j \notin \mathcal{I}$, sample $(ek_j, dk_j) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$, $(\mathsf{vk}_j, sk_j) \leftarrow\!\!\$\ \mathsf{Gen}'(1^\lambda)$, $\omega_j \leftarrow\!\!\$\ \{0,1\}^*$, and invoke $\mathsf{Broadcast}(ek_j || \mathsf{vk}_j || j)$.
3. For each $j \notin \mathcal{I}$, keep running $\mathsf{st}_j \leftarrow\!\!\$\ \mathsf{UpdateState}(1^\lambda)$ and $\mathcal{B}_j \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_j)$ until all the messages $(ek_i, \mathsf{vk}_i)_{i\in[n]} \in \mathcal{B}_j$. Set $\tau^{(0)}_j := (ek_i, \mathsf{vk}_i)_{i\in[n]}$ and $\alpha_j := \tau^{(0)}$.
4. For each $j \notin \mathcal{I}$, emulate the behavior of party P_j as follows:
   (a) Update the state st_j by running $\mathsf{UpdateState}(1^\lambda)$, and let $\mathcal{B}_j \leftarrow\!\!\$\ \mathsf{GetRecords}(\mathsf{st}_j)$.
   (b) Let $\tilde{r} \geq 0$ be the maximum value such that the partial transcript $\tau^{(\tilde{r})} \in \mathcal{B}_j$. Then:
   - If the ciphertexts in $\tau^{(\tilde{r})}$ are not consistent with those in α_j, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate.
   - Else, go to the next step and if α_j is a prefix of $\tau^{(\tilde{r})}$ let $\alpha_j := \tau^{(\tilde{r})}$.

(c) Extract from $\tau_j^{(\tilde{r})}$ the ciphertexts $(c_{i,j}^{(\tilde{r})})_{i\in\mathcal{I}}$ and the signatures $(\sigma_{i,j}^{(\tilde{r})})_{i\in\mathcal{I}}$ that A^* (on behalf of each corrupted player P_i) forwards to P_j. If there exists $i \in \mathcal{I}$ such that $\mathsf{Verify}(\mathsf{vk}_i, \sigma_{i,j}^{(\tilde{r})}) = 0$, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate. Else, for each $r \leq \tilde{r}$, decrypt the ciphertexts $c_{i,j}^{(r)}$ using the decryption key dk_j, and pass the corresponding messages $((m_{i,j}^{(1)})_{i\in\mathcal{I},j\in\mathcal{H}}, \ldots, (m_{i,j}^{(\tilde{r})})_{i\in\mathcal{I},j\in\mathcal{H}})$ to S. Hence:

- Upon receiving `abort` from S, send `abort` to the trusted party, simulate A^* aborting in the real protocol, and terminate.
- Upon receiving $(x_i)_{i\in\mathcal{I}}$ from S, send $(x_i)_{i\in\mathcal{I}}$ to the trusted party, obtain the outputs $(y_i)_{i\in\mathcal{I}}$, and forward $(y_i)_{i\in\mathcal{I}}$ to S. In case S replies with $(\texttt{continue}, \mathcal{H}')$, send $(\texttt{continue}, \mathcal{H}')$ to the trusted party and terminate.
- Upon receiving a set of messages $(m_{j,i}^{(\tilde{r}+1)})_{j\in\mathcal{H},i\in\mathcal{I}}$—corresponding to the simulated messages that each honest player P_j sends to the corrupted party P_i— for each $j \in \mathcal{H}$ and $i \in \mathcal{I}$ compute $c_{j,i}^{(\tilde{r}+1)} \leftarrow\!\!\$\ \mathsf{Enc}(ek_i, , m_{j,i}^{(\tilde{r}+1)})$ (using coins from ω_j) and $\sigma_{j,i}^{(\tilde{r}+1)} = \mathsf{Sign}(sk_j, c_{j,i}^{(\tilde{r}+1)})$. Then, for each $j, j' \in \mathcal{H}$, let $c_{j,j'}^{(\tilde{r}+1)} \leftarrow\!\!\$\ \mathsf{Enc}(ek_{j'}, 0^{|m_{j,j'}^{(\tilde{r}+1)}|})$ (using coins from ω_j) and $\sigma_{j,j'}^{(\tilde{r}+1)} \leftarrow\!\!\$\ \mathsf{Sign}(sk_j, c_{j,j'}^{(\tilde{r}+1)})$, and finally invoke $\mathsf{Broadcast}((c_{j,i}^{(\tilde{r}+1)}||\sigma_{j,i}^{(\tilde{r}+1)})_{i\in[n]\setminus\{j\}})$.

To conclude the proof, we consider a sequence of hybrid experiments (ending with the real experiment) and argue that each pair of hybrids is computationally close thanks to the properties of the underlying cryptographic primitives.

Hybrid $H_3(\lambda)$**:** This experiment is identical to $IDEAL^{\Gamma}_{f_\perp,\mathsf{S}^*,\mathsf{D}^*}(\lambda)$.

Hybrid $H_2(\lambda)$**:** Identical to $H_3(\lambda)$ except that we replace the ciphertexts $(c_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ that each honest party P_j sends to the other honest players $\mathsf{P}_{j'}$ with an encryption of the real messages $(m_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ that the same parties would send in a real execution of π. Note that the other ciphertexts $(c_{j,i}^{(r)})_{j\in\mathcal{H},i\in\mathcal{I}}$ are still emulated using the simulator, and the output of the experiment is determined by the trusted party.

The inputs for the honest parties are chosen to be the values $(x_i)_{i\in\mathcal{H}}$ chosen by the distinguisher D^* at the beginning of the experiment, and the random tape of each player is chosen uniformly once and for all as in the real world.

Hybrid $H_1(\lambda)$**:** Identical to $H_2(\lambda)$ except that we artificially abort if A^* modifies one of the ciphertexts $(c_{j,i}^{(r)})_{j\in\mathcal{H},i\in[n]\setminus\{j\}}$ corresponding to the messages that each honest player sends in a given round. Note that these ciphertexts correspond to both the real messages $(m_{j,j'}^{(r)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ and the simulated messages $(m_{j,i}^{(r)})_{j\in\mathcal{H},i\in\mathcal{I}}$.

Hybrid $H_0(\lambda)$**:** This experiment is identical to $REAL^{\Gamma,0}_{\pi^*,\mathsf{A}^*,\mathsf{D}^*}(\lambda)$.

Lemma 1. $\{H_3(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{H_2(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. We reduce to semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Let $h = |\mathcal{H}|$. For $k \in [0, h]$, consider the hybrid experiment $H_{3,k}(\lambda)$ in which the distribution of the ciphertexts $(c_{j,j'}^{(r+1)})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ is modified as in $H_2(\lambda)$ only for the first h honest parties. Clearly, $\{H_{3,0}(\lambda)\}_{\lambda\in\mathbb{N}} \equiv \{H_3(\lambda)\}_{\lambda\in\mathbb{N}}$ and $\{H_{3,h}(\lambda)\}_{\lambda\in\mathbb{N}} \equiv \{H_2(\lambda)\}_{\lambda\in\mathbb{N}}$.

Next, we prove that for every $k \in [0, h]$ it holds that $\{H_{3,k}(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{H_{3,k+1}(\lambda)\}_{\lambda\in\mathbb{N}}$ which concludes the proof of the lemma. By contradiction, assume that there exists an index $k \in [0, h]$, and a pair of PPT algorithms $(\mathsf{D}^*, \mathsf{A}^*)$ such that D^* can distinguish the two experiments $H_{3,k}(\lambda)$ and $H_{3,k+1}(\lambda)$ with non-negligible probability. We construct a PPT attacker B breaking semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ as follows:

- Receive the target public encryption key ek^* from the challenger.

- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i\in[n]}$, and the auxiliary input z, then pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ to A^*.
- Interact with A^* as described in the ideal experiment, except that:
  - The public encryption key for player P_{k+1} is set to be the target public key ek^*.
  - For each $j \leq k$, when it comes to simulating the ciphertexts $(c^{(r)}_{j,j'})_{j'\in\mathcal{H}\setminus\{j\}}$, use the real messages $(m^{(r)}_{j,j'})_{j'\in\mathcal{H}\setminus\{j\}}$, encrypt them using the public encryption key $ek_{j'}$ of $\mathsf{P}_{j'}$, and sign the ciphertexts with the secret key sk_j (which is known to the reduction).
  - When it comes to simulating the ciphertexts $(c^{(r)}_{k+1,j'})_{j'\in\mathcal{H}\setminus\{j\}}$, forward the pair of plaintexts $(m^{(r)}_{k+1,j'}, 0^{|m^{(r)}_{k+1,j'}|})_{j'\in\mathcal{H}\setminus\{k+1\}}$ to the left-or-right encryption oracle and sign the corresponding ciphertexts using the secret signing key sk_{k+1} of P_{k+1} (which is known to the reduction).
  - For each $j > k+1$, when it comes to simulating the ciphertexts $(c^{(r)}_{j,j'})_{j'\in\mathcal{H}\setminus\{j\}}$, use the dummy messages $(0^{|m^{(r)}_{j,j'}|})_{j'\in\mathcal{H}\setminus\{j\}}$, encrypt them using the public encryption key $ek_{j'}$ of $\mathsf{P}_{j'}$, and sign the ciphertexts with the secret key sk_j (which is known to the reduction).
- Finally, run D^* upon the final output generated by A^*, and return whatever D^* outputs.

Note that the reduction can indeed simulate the interaction with the blockchain protocol Γ as in the ideal experiment, and moreover it can generate the real messages $(m^r_{j,j'})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ as it knows the parties' inputs $(x_i)_{i\in[n]}$. By inspection, the simulation performed by B is perfect in the sense that when the challenger encrypts the messages $m^{(r)}_{k+1,j'}$ the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $H_{3,k+1}(\lambda)$. Similarly, when the challenger encrypts the dummy messages $0^{|m^{(r)}_{k+1,j'}|}$ the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in $H_{3,k}(\lambda)$. Hence, B breaks semantic security of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with non-negligible probability, concluding the proof. □

Lemma 2. $\{H_2(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{H_1(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. Let BAD be the event that an artificial abort happens in $H_1(\lambda)$. Note that this means that, for some $j \in \mathcal{H}$, the attacker A^* replaces one of the ciphertexts $c^{(r)}_{j,i}$ that P_j would send to P_i in the real protocol with a different ciphertext $\tilde{c}^{(r)}_{j,i}$, in such a way that the corresponding signature $\tilde{\sigma}^{(r)}_{j,i}$ is still accepting. Clearly, the experiments $H_2(\lambda)$ and $H_1(\lambda)$ are identical conditioning on BAD not happening, and does it suffices to show that $\mathbb{P}[BAD]$ is negligible.

Given a PPT distinguisher D^* and a PPT attacker A^* such that A^* provokes event BAD in a run of $H_2(\lambda)$ with non-negligible probability, we can construct a PPT attacker B breaking security of the signature scheme $(\mathsf{Gen}', \mathsf{Sign}, \mathsf{Verify})$. The reduction works as follows:

- Receive the target public verification key vk^* from the challenger.
- Choose a random j^* as a guess for the index corresponding to the honest party for which A^* provokes the bad event.
- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i\in[n]}$, and the auxiliary input z, then pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ to A^*.
- Interact with A^* as described in $H_2(\lambda)$, except that:
  - The public verification key for player P_{j^*} is set to be the target public key vk^*.
  - When it comes to simulating the round-r messages from party P_{j^*}, generate the ciphertexts $(c^{(r)}_{j^*,i})_{i\in[n]\setminus\{j^*\}}$ as done in $H_2(\lambda)$, and then forward each of $c^{(r)}_{j^*,i}$ to the challenger, obtaining the corresponding signature $\sigma^{(r)}_{j^*,i}$ that is needed in order to complete the simulation.

  - Keep updating the local state of P_{j^*} until an index $i \in [n] \setminus \{j^*\}$ is found such that the partial transcript α_{j^*} contains a pair $(\tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i})$ such that $\mathsf{Verify}(\mathsf{vk}_{j^*}, \tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i}) = 1$ and $\tilde{c}^{(r)}_{j^*,i}$ is different from the original ciphertext $c^{(r)}_{j^*,i}$ previously sent on behalf of P_{j^*}.

- If no such pair is found, abort the simulation. Else, return $(\tilde{c}^{(r)}_{j^*,i}, \tilde{\sigma}^{(r)}_{j^*,i})$.

Note that the simulation performed by B is perfect, in that the view of $(\mathsf{D}^*, \mathsf{A}^*)$ is identical to that in a run of $H_2(\lambda)$. Moreover, conditioning on B guessing the index j^* correctly, the reduction is successful in breaking security of the signature scheme exactly with probability at least $\mathbb{P}[BAD]$, which is non-negligible. Since the former event also happens with non-negligible probability, this concludes the proof. □

Lemma 3. $\{H_1(\lambda)\}_{\lambda\in\mathbb{N}} \approx_c \{H_0(\lambda)\}_{\lambda\in\mathbb{N}}$.

Proof. The proof is by reduction to UC-security of the underlying protocol π. By contradiction, assume that there exists a PPT adversary A^* and a non-uniform PPT distinguisher D^* that can distinguish between $H_1(\lambda)$ and $H_0(\lambda)$ with non-negligible probability. Consider the following PPT attacker A, initialized with a set of corrupted parties $\mathcal{I}$, inputs $(x_i)_{i\in\mathcal{I}}$ for the malicious players, and auxiliary input $z = (z^*, (ek_i, dk_i)_{i\in[n]}, (\mathsf{vk}_i, sk_i)_{i\in[n]})$ which will be specified later:

- Pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z^*)$ to A^*.
- For each $i \in \mathcal{I}$, upon receiving the round-r messages $(m^{(r)}_{j,i})_{j\in\mathcal{H}}$ from the honest players to the malicious players, let $c^{(r)}_{j,i} \leftarrow\!\!\$\ \mathsf{Enc}(ek_i, m^{(r)}_{j,i})$ and $\sigma^{(r)}_{j,i} = \mathsf{Sign}(sk_j, c^{(r)}_{j,i})$, and emulate broadcasting $(c^{(r)}_{j,i}, \sigma^{(r)}_{j,i})_{j\in\mathcal{H},i\in\mathcal{I}}$ via the blockchain protocol.
- For each $j \in \mathcal{H}$, upon receiving the round-r messages $(m^{(r)}_{i,j})_{i\in\mathcal{I}}$ that A^* wants to send to the honest parties, let $c^{(r)}_{i,j} \leftarrow\!\!\$\ \mathsf{Enc}(ek_j, m^{(r)}_{i,j})$ and $\sigma^{(r)}_{i,j} = \mathsf{Sign}(sk_i, c^{(r)}_{i,j})$, and emulate broadcasting $(c^{(r)}_{i,j}, \sigma^{(r)}_{i,j})_{i\in\mathcal{I},j\in\mathcal{H}}$ via the blockchain protocol.
- For each $j \in \mathcal{H}$, compute the messages $(m^{(r)}_{j,j'})_{j'\in\mathcal{H}\setminus\{j\}}$ exchanged between honest parties as done in H_0 (which is the same in $H_1(\lambda)$), let $c^{(r)}_{j,j'} \leftarrow\!\!\$\ \mathsf{Enc}(ek_{j'}, m^{(r)}_{j,j'})$ and $\sigma^{(r)}_{j,j'} = \mathsf{Sign}(sk_j, c^{(r)}_{j,j'})$, and emulate broadcasting $(c^{(r)}_{j,j'}, \sigma^{(r)}_{j,j'})_{j\in\mathcal{H},j'\in\mathcal{H}\setminus\{j\}}$ via the blockchain protocol.
- In case a fork appears during the simulation of the underlying blockchain protocol, replicate the messages from the honest players as done in the other branches (using exactly the same randomness). On the other hand, if the messages from A^* differ from those sent on the simulation of a previous branch, simulate A^* aborting and terminate.
- Output whatever A^* outputs.

Additionally, let Z be the following PPT distinguisher:

- Run D^*, receiving the set of corrupted parties $\mathcal{I}$, the inputs $(x_i)_{i\in[n]}$, and the auxiliary input z^*, then sample (ek_i, dk_i) and (vk_i, sk_i) for all $i \in [n]$, and pass $(\mathcal{I}, (x_i)_{i\in\mathcal{I}}, z)$ to the above defined attacker A, where $z = (z^*, (ek_i, dk_i)_{i\in[n]}, (\mathsf{vk}_i, sk_i)_{i\in[n]})$.
- Upon receiving the final output from A, pass it to D^* and output whatever D^* outputs.

By inspection, in case the attacker A is playing in a real execution of protocol π, the view of D^* is identical to that in an execution of $H_0(\lambda)$ with A^* controlling the malicious parties. Similarly, in case the view of A is emulated using the simulator S (corrupting the dummy parties controlled by A) of protocol π, the view of D^* is identical to that in an execution of $H_1(\lambda)$ with A^* controlling the malicious parties. It follows that Z can distinguish between $REAL_{\pi,\mathsf{A},\mathsf{Z}}(\lambda)$ and $IDEAL_{f_\perp,\mathsf{S},\mathsf{Z}}(\lambda)$ with non-negligible probability, a contradiction. □

The theorem now follows directly by combining the above lemmas. □

3.2.3 On Fairness with Penalties

In their work, Andrychowicz *et al.* [ADMM14, ADMM16] proposed a different notion of fairness for MPC protocols that run on blockchain: fairness with penalties. This notion states that if an adversary in an MPC protocol decides to abort the execution of the protocol it will be financially penalized. To obtain the penalization in the lottery protocol, Andrychowicz *et al.* added a deposit step in the protocol.

Our compiler described in §3.2 does not suffer we discuss here how to obtain fairness with penalties following in part the outline of [KB14, BK14b].

Let us now assume the existence of an (n, n)-secret sharing scheme $(\mathsf{Share}, \mathsf{Recon})$, non-interactive commitment schemes, and consider a functionality f' that first calculates $y \leftarrow f(x_1, \ldots, x_n)$, where each party P_i holds x_i, and then calculates the shares of the output $(\sigma_1, \ldots, \sigma_n) \leftarrow\!\!\$\ \mathsf{Share}(y)$, the commitments $C = (\gamma_1, \ldots, \gamma_n)$, where $\gamma_i \leftarrow\!\!\$\ \mathsf{Com}(\sigma_i)$, and outputs (C, σ_i) to each player P_i. Let us call π' the protocol realizing f'. We can apply our generic compiler to π' to obtain a protocol π'_{bc} that can be run in the blockchain. Our protocol π_{fair}, running with players $\mathsf{P}_1 \ldots, \mathsf{P}_n$ works as follows

(i) *Protocol execution*: All the players engage in the protocol π'_{bc}. A party P_i aborts the execution if π'_{bc} aborts. Otherwise, obtains (C, σ_i) in the last round.

(ii) *Smart contract*: P_1 publishes the smart contract depicted in Fig. 3.2.

(iii) *Commitment phase*: For each $i \in [n]$, P_i triggers $\mathsf{deposit}(C_i)$ together with d coins, where d is a fixed deposit. If some player does not publish his commitments with the deposit or there is a disagreement on the commitments within time1 (i.e., a player P_j sends $C_j \neq C_i$ for some $P_{i \neq j}$), or deposits a value $d_i < d$, P_i abort the execution. Recall that abort in this phase is still fine, since no information about the output y is released. Otherwise, if time1 has passed, go to the Opening Phase.

(iv) *Opening phase*: For each $i \in [n]$, P_i opens his commitment by sending $\mathsf{openCom}(i, \sigma_i)$, thus receiving back his d coins, wait that all the openings are published in the smart contract (within time2) and calculates $y \leftarrow \mathsf{Recon}(\sigma_1, \ldots, \sigma_n)$. If, after time2, some share is missing, P_i aborts the execution.

During the last phase, if some player did not open the commitment or sent an incorrect value, the smart contract will penalize him by freezing his deposit. Thus, the adversary is not incentivized to send an incorrect share.

This attempt to add fairness with penalties, however, introduces an attack. Given an n-party protocol $\pi^{\Gamma}_{f'}$ obtained by the compiler described in §3.2 applied to $\pi_{f'}$, with the addition of the smart contract, commit and opening phases described above, we have the following scenario:

- For all $i \in [n]$, party P_i runs π_{fair} obtaining (C, σ_i).
- For all $i \in [n]$, party P_i triggers $\mathsf{deposit}(C)$ together with d coins to the smart contract.
- For all $i \in [n-1]$, party P_i opens his commitment by triggering $\mathsf{openCom}(\sigma_i)$.
- Wlog., we say that P_n is an adversary. P_n computes the output y. If P_n does not like y in the current branch, P_n can try to exploit a fork happening during the execution of π_{fair} to change the in a different branch to obtain a new couple (C', σ'_n).
- The honest parties $\mathsf{P}_1, \ldots, \mathsf{P}_{n-1}$ notice that there is a message published by P_n that differs from the value previously received by P_n. Since the transcript obtained from the blockchain differs from the transcript stored in their local state, they will abort.

The **General Compiler Smart Contract** runs with players $\mathsf{P}_1, \ldots, \mathsf{P}_n$ and consists of two main functions deposit and openCom and two fixed timestamps time1, time2.

Commitment Phase: In round t_1, when $\mathsf{deposit}(C_i)$ together with d coins is triggered from a party P_i, store (i, C_i). Then, if $\forall i, j : C_i = C_j$ proceed to the Opening Phase. Otherwise, for all i, if the message (i, C_i) has been stored, send message d coins to P_i and terminate.

Opening Phase: In round t_2, when $\mathsf{openCom}(i, \sigma_i)$ is triggered from P_i, check if $\mathsf{Com}(\sigma_i) = \gamma_i$, where γ_i is obtained from parsing $C_i = (\gamma_1, \ldots, \gamma_n)$ (recall that all the C_i are the same), and send d coins back to P_i.

Figure 3.2: General compiler smart contract.

The protocol described is not fair, since we can construct a counterexample that proves that the unfair party P_n can obtain the output without being penalized.

We now state the issue more formally. $\mathsf{P}_1, \ldots, \mathsf{P}_n$ will play $\pi_{f'}^{\Gamma}$ to obtain (C, σ_i). As the smart contract is published, P_n will trigger $\mathsf{deposit}(C)$ together with d coins. At this point, P_n waits that all P_i, with $i \in [n]$ publish the opening σ_i.

When P_n sees $\sigma_1, \ldots, \sigma_{n-1}$ on the blockchain, it computes the output. If P_n dislikes the output, then he tries to exploit a branch created before the end of the execution of π_{fair} to change messages in that branch to obtain an advantage. Since P_n publishes different messages on different branches of the blockchain, there exist some party P_i, with $i \in [n]$ that will notice it, causing an abort in the protocol.

Let's call b_1 the branch where P_n learned the output and b_2 the branch exploited to change the execution of π_{fair}. We have two cases:

- If b_1 is the branch that will be confirmed on the blockchain, P_n will be penalized.
- If b_2 is the branch that will be confirmed on the blockchain, P_n will cause an abort in the protocol before that the commitment phase starts. In this case he does not get penalized for learning the output.

With this counterexample we show that the proposed solution is not enough to obtain fairness with penalties, since P_n has the possibility to learn the output without incurring in any punishment. It is possible to obtain fairness with penalties using our general compiler (see next), and waiting that the commitment phase is confirmed on the ledger. Indeed, if the commitment phase described in Figure 3.2 is finalized on the blockchain, the adversary A cannot change the commitment done on the blockchain without losing her deposit. A is forced to continue the run wit the finalized commitment phase to receive back the deposit.

More precisely, an adversary A cannot learn y unless A decides to lose the d coins deposited in the commitment phase. Since the commitment phase is confirmed on the ledger, A cannot find a fork to exploit the execution of the protocol on another branch. Yet, A can cause an abort in the protocol, but if it happens before the commitment phase she will not learn the output y. If the abort happens after the commitment phase, A will learn the output but will be penalized.

Theorem 4. *Let's assume the existence of non-interactive commitment schemes and (n, n)-secret sharing schemes. Let π'_{bc} be an n-party ρ-round protocol realizing f' in the presence of quick players. Then, the protocol π_{fair} described above securely realizes f satisfying fairness with penalties in the presence of $(\rho, 1)$-quick players.*

Proof Sketch. We can claim security of the the compiled protocol π'_{bc} obtained by applying the general compiler to π', by referring to the same proof of Theorem. 3. Now, we argue that the overall protocol π_{fair} achieves fairness with penalties. As mentioned before, aborts during the execution of π'_{bc} are acceptable, since the adversary cannot

learn any information about the output. After the committing phase, that is finalized, the adversary could try to exploit different branches to send different openings of his commitments. We have the following time-line: The execution started in a branch b_1 and a forks happens right after the committing phase, generating a branch b_2. Wlog. of generality we can extend this argument to multiple parallel executions in different branches. We have the following scenarios:

Scenario 1: A corrupted player abort in both branches. Since the commitments are finalized, fairness with penalties follows in a straightforward manner, since he did not open his commitments in each branch, and so also in the confirmed one.

Scenario 2: A corrupted player opens his share in b_1 and aborts the execution in b_2 (after the commitment phase). If b_1 gets confirmed, the honest parties will learn the output. If b_2 gets confirmed, it automatically boils down to Scenario 1.

Scenario 3: A corrupted player P_i opens his share in b_1 and tries to open on a different share in b_2. Since the commitment is always confirmed, the adversary cannot try to change his commitment by exploiting forks. If A is able to open the commitment by providing two different shares, then we can define an adversary $\mathsf{A}_{\mathsf{com}}$ breaking the binding property of the underlying commitment scheme with non-negligible probability. That means that at least in one of the two branches P_i gets penalized, and if he provides the correct opening in one of the branches and it gets confirmed, honest players will learn the output.

□

Chapter 4

Privacy-Preserving Auditable Token Payments in a Permissioned Blockchain System

Token management systems were the first application of blockchain technology and are still the most widely used one. Early implementations such as Bitcoin or Ethereum provide virtually no privacy beyond basic pseudonymity: all transactions are written in plain to the blockchain, which makes them perfectly linkable and traceable.

Several more recent blockchain systems, such as Monero or Zerocash, implement improved levels of privacy. Most of these systems target the *permissionless* setting, just like Bitcoin. Many practical scenarios, in contrast, require token systems to be *permissioned*, binding the tokens to user identities instead of pseudonymous addresses, and also requiring auditing functionality in order to satisfy regulation such as AML/KYC.

We present a privacy-preserving token management system that is designed for permissioned blockchain systems and supports fine-grained auditing. The scheme is secure under computational assumptions in bilinear groups, in the random-oracle model.

4.1 Introduction

4.1.1 Motivation

Early implementations of token payment systems such as Bitcoin or Ethereum provide virtually no privacy beyond basic pseudonymity: all transactions are written in plain to the blockchain, which makes them linkable and traceable.

Several approaches exist for adding different levels of privacy to blockchain-based transactions. *Tumblers* such as CoinJoin [Max13] combine several transactions of different users and obscure the relation between payers and payees. In *mix-in*-based systems such as CryptoNote [vS13], transactions reference multiple superfluous payers that do however not actually participate in the transaction and only serve as a cover-up for the actual payer. Confidential Assets [PBF+18] hide the amounts in a payment but leave the payer-payee relation in the open. Finally, advanced systems such as Zerocash [BSCG+14] both encrypt the amounts and fully hide the payer-payee relation.

While the privacy of transactions is important, it should not void the requirements of transparency and auditability, especially in permissioned networks that come with strong identity management and promise to ensure accountability and non-deniability. This chapter introduces a solution dedicated to the permissioned setting to cover this gap: it hides the content of transactions without preventing authorized parties from auditing them.

Another goal of this chapter is to move away from complex and non-falsifiable computational assumptions that underpin zkSNARK-based schemes and instead work with more conservative assumptions. Restricting

ourselves to the permissioned setting allows us to leverage a combination of signatures and standard ZK-proofs to achieve these goals.

4.1.2 Related Work

Various solutions for improving privacy in blockchain-based token systems exist. We briefly review the most related ones.

Miers et al. [MGGR13] introduced Zerocoin, which allows users to anonymize their bitcoins by converting them into *zerocoins* that rely on Pedersen commitments and zero-knowledge proofs. Zerocoins can be changed back to bitcoins without leaking their origin. Zerocoin however does not offer any transacting or auditing capabilities.

Confidential Assets of Poelstra et al. [PBF+18] protect privacy (in a limited form) by hiding the types and the values of the traded assets. The idea, similarly to Zerocoin, is to use Pedersen commitments to encode the amount and types of traded assets, and zero-knowledge proofs to show the validity of a transaction. The proposed scheme however does not hide the transaction graph or the public keys of the transactors. While this allows for some form of public auditability, it hinders the privacy of the transacting parties.

Zerocash [BSCG+14] is the first fully anonymous decentralized payment scheme. It offers unconditional anonymity, to the extent that users can repudiate their participation in a transaction. Thanks to a combination of hash-based commitments and zkSNARKs, Zerocash validates payments and prevents double-spending relatively efficiently. On the downside, Zerocash requires a trusted setup and an expensive transaction generation and its security relies on non-falsifiable assumptions.

Extensions to Zerocash have been proposed [GGM16] to support expressive validity rules to provide accountability: notably, the proposed solution ensures regulatory closure (i.e. allowing exchanges of assets of the same type only) and enforcing spending limits. In terms of accountability, the proposed scheme allows the tracing of certain *tainted* coins, while not really extensively and consistently allowing transactions to be audited. By building on Zerocash, the proposed scheme inherits the same limitations regarding computational assumptions and trusted setup.

QuisQuis [FMMO18] and Zether [BAZB19] propose solutions that provide partial anonymity. On a high level, instead of sending a transaction that refers only to the accounts of the sender and the recipients of a payment, the sender adds accounts of other users, who act as an anonymity set (similar to CryptoNote [vS13]). Both schemes couple ElGamal encryption with Schnorr zero-knowledge proofs to ensure that user accounts reflect the correct payment flows. Contrary to Zerocash, QuisQuis and Zether rely on falsifiable assumptions and do not require any trusted setup.

Solidus [CZJ+17] is a privacy-preserving protocol for asset transfer that is suitable for intermediated bilateral transactions, where banks act as mediators. Solidus conceals the transaction graph and values by using banks as proxies. The authors leverage ORAMs to allow banks to update the accounts of their clients without revealing exactly which accounts are being updated. The novelty of Solidus is PVORM (Publicly Verifiable Oblivious RAM Machine), which is an ORAM that comes with zero-knowledge proofs that show that the ORAM updates are correct with respect to the transaction triggering them. In Solidus there is no dedicated auditing functionality; however banks could open the content of relevant transactions at the behest of authorized auditors.

The zkLedger protocol of Narula, Vasquez, and Virza [NVV18] is a permissioned asset transfer scheme that hides transaction amounts as well as the payer-payee relationship and supports auditing. One main difference with our approach is the end user: zkLedger aims at a setting where the transacting parties are banks, whereas our solution considers the end user to be the client of "a bank." This is why zkLedger enjoys relatively more efficient proofs and could afford a transaction size that grows linearly with the number of total transactors in the platform (i.e. banks), which is inherently small. (In our scheme, transaction sizes do *not* grow with the number of overall parties.) Similarly when it comes to auditability, zkLedger offers richer and more flexible semantics but at the expense of audit granularity. Auditing in zkLedger is limited to banks and does not cover cases where auditors are required to monitor the transaction flow of the clients (of the banks).

4.1.3 Results

We describe a token management system for permissioned networks that enjoys the following properties:

Privacy: Transactions written on the blockchain conceal both the values that are transferred and the payer-payee relationship. The transaction leaks no information about the tokens spent in this transaction beyond the fact that they are valid and unspent.

Authorization: Users authorize transactions via credentials; i.e., the authorization for spending a token is bound to the user's identity instead of a pseudonym (or address). The authorization makes use of anonymous credentials and is privacy-preserving.

Auditability: Each user has an assigned auditor that is allowed to see the transaction information *related to that particular user*.

Satisfying these three requirements is crucial for implementing a payment system that protects the users' privacy but at the same time complies with regulation.

The system we propose is based on the unspent transaction output (UTXO) model pioneered by Bitcoin [Nak08b] and supports multi-input-multi-output transactions. It inherits several ideas from prior work, such as the use of Pedersen commitments from Confidential Assets [PBF+18] and the use of serial numbers to prevent double-spending from Zerocash [BSCG+14]. These are combined with a blind certification mechanism that guarantees the validity of tokens via threshold signatures, and with an auditing mechanism that allows flexible and fine-grained assignment of users to auditors.

We use a selection of cryptographic schemes that are based in the discrete-logarithm or pairing settings and are structure-preserving, such as Dodis-Yampolskiy VRF [DY05], ElGamal encryption [ElG85a], Groth signatures [Gro15], Pedersen commitments [Ped91b], and Pointcheval-Sanders signatures [PS16]. This allows us to use the relatively efficient Groth-Sahai proofs [GS08] and achieve security under standard assumptions, in the random-oracle model.

Outline. The remainder of the chapter is structured as follows. In Section 4.2, we provide further background on several important techniques. Section 4.3 then shows an overview of our protocol. Section 4.4 describes the types of cryptographic schemes used in the protocol, before Section 4.5 specifies the security model. Section 4.6.5, contains the protocol description and the security analysis. Section 4.7 provides details on how to instantiate the protocol using well-established primitives that do not require any complex setup assumptions. In Section 4.8, we describe the implementation and the performance measurements.

4.2 Background

4.2.1 Decentralized token systems

A decentralized token transfer is performed by appending a `transfer` transaction to the blockchain. Such a transaction comprises the transfer details (e.g. sender, receivers, type and value) and a proof that the author of the transaction possesses enough liquidity to perform the transfer. The transaction is then validated against the blockchain state (i.e. the ledger). More precisely, the blockchain checks that the origin of the transaction has the right to transfer the token and that the overall quantity of tokens is preserved during the transfer. Existing decentralized token systems are either *account-based* (e.g. [Eth]) or *unspent transaction outputs (UTXO)-based* (cf. [Nak08b]). A valid transfer in an account-based systems results in updating the accounts of the sender and the receivers. In a UTXO-based token system, a `transfer` transaction includes a set of inputs—tokens to be consumed—and outputs—tokens to be created. A valid transfer in such systems leads to destroying the inputs and adding the outputs to the ledger to be later consumed by subsequent transactions.

4.2.2 Privacy-preserving token systems

Decentralization of token systems gives rise to serious privacy threats: if transactions contain the transfer information in the clear, then anyone with access to the ledger is able to learn the history of each party's transaction. We call a decentralized token system *privacy-preserving* if it partially or fully hides the transfer details. Examples of decentralized and privacy preserving token systems are Confidential Assets [PBF+18], Zether [BAZB19], QuisQuis [FMMO18] and Zerocash [BSCG+14], with the last one offering the highest level of privacy protection.

Zerocash. The privacy in Zerocash relies on a combination of commitments, zkSNARKs and Merkle-tree membership proofs. Namely, tokens in Zerocash are computed as a hiding commitment to a value, a type and an owner's pseudonym. After its creation, a token is added to a *public Merkle tree* and during a transfer, the origin of the transaction proves in zero-knowledge that the token is valid (i.e. included in the Merkle tree), that it was not spent before and that she owns it. Thanks to zkSNARKs, transaction validation in Zerocash is quite fast. Yet, this comes at the cost of a *complex trusted setup* and a very expensive proof generation. To obviate these two limitations, we exploit the properties of *permissioned token systems* to replace Merkle trees with signature-based membership proofs, in order to devise a solution that relies only on Groth-Sahai proofs [GS08].

4.2.3 Permissioned token systems

In a permissioned token system such as Hyperledger Fabric [ABB+18] or Quorum [quo], a user is endowed with a long-term credential that reflects her attributes and role. Tokens are introduced by special users, called *issuers*, through `issue` transactions. These transactions are then validated against predefined policies that reflect existing norms and regulations. For example, issuing policies define which issuers are authorized to create which tokens and under which conditions. Similarly to `issue` transactions, `transfer` can also be validated against policies: the simplest of which is that a transfer can take place only between registered users. A fundamental property of permissioned systems is that transactions are signed using long-term credentials. As a by-product, transactions can be traced back to their origin, enforcing thus the requirements of auditability and accountability.

4.2.4 Signature-based membership proofs

We use signatures to implement zero-knowledge membership proofs [CCas08]. Roughly speaking, consider a set $\mathbb{S}$ that consists of elements that are signed using a secret key sk associated with $\mathbb{S}$. It follows that proving knowledge of some e in $\mathbb{S}$ in zero-knowledge amounts to (i) computing a hiding commitment of e; (ii) and then proving knowledge of a signature, computed with sk, on the committed value. In this paper, we use this mechanism for two purposes: (i) to prove that a user is in the set of registered users; (ii) and to show that a token is in the set of *valid* tokens recorded in the ledger.

4.2.5 Encryption-based auditability

In an encryption-based auditable token system, transactions carry ciphertexts intended for the authorized auditors. For such a mechanism to be viable, it is important to ensure that (i) the ciphertexts encrypt the correct information; (ii) and they are computed using the correct keys. This can be achieved through zero-knowledge proofs—computed by the creator of the transaction—that link the ciphertexts to the transfer details and attest that the two requirements listed above are not violated.

4.3 Overview

4.3.1 Design Approach

The first component of our solution is *token encoding*. Each token is represented by a hiding commitment (e.g. Pedersen's) that contains the identifier of the token owner, the value of the token and its type. The life-cycle of

a token is governed by two transactions: `issue` and `transfer`. An `issue` transaction creates a token of a given type and value and assigns it to the issuer (i.e. author of the transaction). For an `issue` transaction to be valid it should be submitted by the authorized issuer. For ease of exposition, we assume that a token issuer can issue only one type of tokens and we conflate the type of a token with its issuer. Once a token is created it changes ownership through `transfer` transaction. Given that we operate within the UTXO framework, `transfer` transaction consists of a set of input tokens to be consumed and a set of output tokens to be created and it is validated against the following rules:

- The author of the transaction is the rightful owner of the input tokens;
- the owners of the output tokens are registered;
- the type and the value of tokens are preserved;
- the input tokens can be traced back to *valid* transactions in the ledger;
- the input tokens were not consumed before (to prevent double spending).

Our solution moves away from zkSNARK and their trusted setup assumption and relies only on standard NIZK proofs (e.g. Groth-Sahai's [GS08]). More precisely, it leverages the *permissioned setting* to use ZK signature-based membership proofs to ascertain that a user is registered and that a token belongs to the ledger in a privacy-preserving manner. Namely, we assume that there is a *registration authority* that provides authorized users with long-term credentials (i.e. signatures) with their attributes, and a *certifier* that a user contacts with a *certification* request to vouch for the validity of tokens she owns. A *certification* request contains a token (i.e. commitment) and upon receiving such a request the certifier checks whether the token is included in a valid transaction in the ledger. If so, the certifier *blindly* signs the token and the resulting signature can be used subsequently to prove that the token is legitimate.

To prevent double spending, we leverage serial numbers to identify tokens when they are consumed, as in Zerocash. It is important that these serial numbers satisfy the following security properties: (i) collision resistance: two tokens result in two different serial numbers; (ii) determinism: the same token always yields the same serial number; (iii) unforgeability: only the owner of the token can produce a valid serial number. We use *verifiable random functions* (e.g. Dodis-Yampolskiy [DY05]) to generate serial numbers that are a function of the token owner secret key and a randomness that is tied to the token at its creation time.

To enable auditability, we encrypt the information in `transfer` transactions (i.e. sender, receivers, types and values) under the public keys of the sender's and the receivers' auditors. To accommodate real-world use-cases, our solution does not assume a single auditor for all users. This means that the encryption scheme must not only be *semantically-secure* but also *key-private*, such as ElGamal.

4.3.2 Architectural Model

Participants

Our solution involves the following types of users:

Users

They own tokens that represent some real-world assets, and wish to exchange their tokens with other users in the network. This is achieved through `transfer` transactions.

Issuers

They are users who are authorized to introduce tokens in the system through `issue` transactions. For simplicity purposes, we assume that each issuer is allowed to introduce only one type of token and that the type of token is defined as the identifier of the respective issuer.

Auditors

These are entities with the authority and responsibility to inspect transactions of users. We assume that each user in the system is assigned an auditor at registration time and that this assignment is immutable.

Certifier

This is a privileged party that provides users with *certificates* that vouch for the validity of their tokens. More specifically, a user who wishes to transfer the ownership of a token contacts the certifier with the token; the certifier in turn inspects whether the token appears in a *valid transaction* in the ledger or not. If so, the certifiers sends a certificate (i.e. signature) to that effect to the user.

Registration authority

This is a privileged party that generates long-term credentials for all the participants in the system, including users, issuers, auditors and certifiers. Namely, the credentials tie the real-world identity of the requestor to her attributes and her public keys. An example of an attribute in our solution is the role (e.g., "user", "auditor", "certifier") that determines what type of credentials to be generated. A *user credential* is a signature that binds the user public keys to both her identifier and her auditor's identifier; whereas an *auditor credential* is a signature that links the auditor's encryption key to her identifier; finally the certifier's credential is a signature of her public key.

Ledger

This is a decentralized data store that keeps records of all `issue` and `transfer` transactions that have been previously submitted. It is accessible to all parties in the system to read from and submit transactions to. The ledger has a *genesis block* that contains (i) the system security parameters; (ii) the public information of the registration authority and the credentials of the certifier; and (iii) the identifiers of the issuers authorized to introduce tokens in the system.

Interactions

The interactions between system participants are shown in Figure 4.1. At first users, issuers, auditors and certifier engage with the registration authority in a *registration* protocol to get long-term credentials for their subsequent interactions.

A genesis block is created that announces the system parameters, the public information of the registration authority, the credentials of the certifiers and an *initial list* of authorized issuers. From now on, the system will be able to accommodate token management requests. More precisely, issuers submit `issue` transactions to the ledger to introduce new tokens, and the ledger ensures that all incoming transactions are correctly stored. Anyone with access to the ledger, in particular the certifier and auditors, can verify whether the transaction is valid or not using the information in the *genesis* block. Subsequently, *token transfer* operations take place between users through `transfer` transactions. The ledger again stores the transaction to make it available to all participants. For simplicity, we assume that the ledger accepts all transactions without verification. However to transfer a token, a user contacts the certifier that checks if the transaction including the token is valid. Only then the certifier signs the token making it transferable. To inspect a user activity, an auditor reads transactions from the ledger, checks their validity and tries to decrypt them with her secret key only if they are valid.

4.3.3 Trust Model

Registration authority. We assume that the registration authority is trusted to assign *correct credentials* to all parties in the system. A participant presents a set of attributes and her public key to the registration authority and

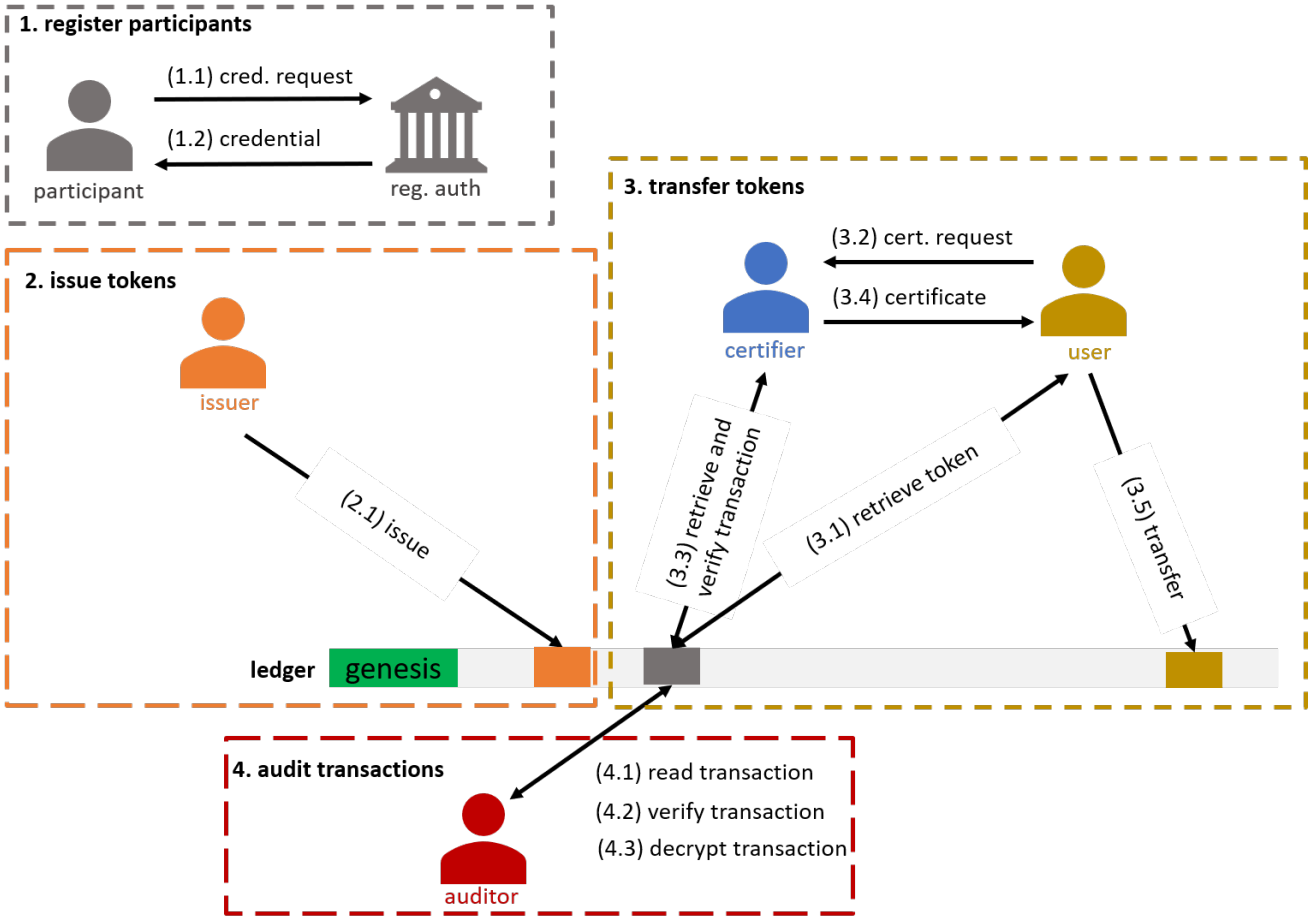

Figure 4.1: This figure shows the interactions between the participants in our system. Users, issuers, auditors and the certifier are granted credentials by interacting with the registration authority. Upon an external decision to create new tokens, one or more issuers submit an `issue` transaction to the ledger and the ledger automatically adds the transaction. To transfer a token, a user contacts the certifier with a certification request that references the token to be signed and the transaction that created it. If it is an `issue` transaction, the certifier checks if the author of the transaction is authorized. If it is a `transfer` transaction, the certifier verifies if the ZK proof is valid. Once the owner of a token receives the corresponding certificate, she can transfer it to registered users. Finally, auditors assigned to a user can audit that user's transactions by obtaining access to the ledger.

receives in return a credential that binds her attributes to her public key. It is incumbent upon the registration authority to verify the correctness of the attributes of a participant prior to sending the credential. For example, it should verify that the participant knows the secret key underlying the advertised public key; it should also verify in the case of users that the announced auditors are legitimate. Furthermore, the registration authority is trusted to assign *one unique credential* per participant. However, it is not *trusted* regarding the privacy of users.

Notice that the trust assumption in the registration authority can be relaxed via a distributed registration protocol.

Users. Users may collude to compromise the security of the system. They may attempt to steal tokens of others, double-spend their own tokens, forge new tokens, transfer tokens to non-registered users, encrypt incorrect information in the auditors' ciphertexts, etc. They may also attempt to undermine the privacy of honest users by de-anonymizing transactors, linking tokens, learning the content of transactions, etc.

Issuers. Issuers are users that are trusted to introduce tokens of a certain type. However, issuers may collude to surreptitiously create tokens on behalf of other honest issuers, compromise the privacy of users and obviate auditing among other things.

Certifier. To transfer a token, a user contacts the certifier to receive a signature that proves the token validity (i.e. inclusion in a valid transaction in the ledger). Accordingly, the certifier is trusted to generate signatures only for tokens that can be traced back to valid transactions in the ledger. We can relax this trust assumption using a *threshold signature* scheme that distributes the certification process and guarantees its integrity as long as the majority of the signers (i.e. certifiers) is honest.

While certifiers may be able to link `transfer` transactions referencing certified tokens to certification requests, they should not be able to derive any further information about the transactions in the ledger or the tokens they certify.

Auditors. Auditors are authorized to only learn the information pertained to their assigned users. That is, colluding users and auditors should not be able to derive any information about the token history of users who are not assigned to the malicious auditors.

Ledger. For simplicity purposes, we use the ledger only as a time-stamping service. It does not perform any transaction validation, rather it stores the full transaction including the proofs of correctness. Anyone later can check the transaction, verify the proofs and decide if the transaction is valid or not. We assume however that the ledger is *live* and *immutable*: a transaction submitted to the ledger will eventually be included and cannot be deleted afterwards.

4.4 Cryptographic Schemes

The section presents the cryptographic schemes that will be used to build the protocol. We only present them briefly, and provide more information on concrete instantiations later in the chapter. All cryptographic algorithms are parameterized by a so-called *security parameter* $\lambda \in \mathbb{N}$ given (sometimes implicitly) to the algorithms.

4.4.1 Commitment Schemes

A commitment scheme COM consists of three algorithms $\mathrm{ccrsgen}$, commit, and open. The *common reference string (CRS)* generator $\mathrm{ccrsgen}$ is probabilistic and, on input the security parameter λ, samples a CRS $crs \leftarrow\!\!\$\ \mathrm{ccrsgen}(\lambda)$. The commitment algorithm is a probabilistic algorithm that, on input of a vector $(m_1, \ldots, m_\ell)$ of messages, outputs a pair $(cm, r_{\mathrm{cm}}) \leftarrow\!\!\$\ \mathrm{commit}(crs, (m_1, \ldots, m_\ell))$ of commitment cm and opening r_{cm}. We

sometimes also use the notation $cm \leftarrow \mathrm{commit}(crs, (m_1, \ldots, m_\ell); r_{\mathrm{cm}})$ to emphasize that a specific random string r_{cm} is used. Finally, there is a deterministic opening algorithm $\mathrm{open}(crs, cm, (m_1, \ldots, m_\ell), r_{\mathrm{cm}})$ that outputs either true or false.

Commitments must be *hiding* in the sense that, without knowledge of r_{cm}, they do not reveal information on the committed messages, and they must be *binding* in the sense that it must be infeasible to find a different set of messages $m'_1, \ldots, m'_\ell$ and r'_{cm} that open the same commitment.

4.4.2 Digital Signature Schemes

A digital signature scheme SIG consists of three algorithms $\mathrm{skeygen}$, sign, and Vf. The key generation algorithm $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathrm{skeygen}(\lambda)$ takes as input the security parameter λ and outputs a pair of private (or secret) key sk and public key pk. Signing algorithm $\mathsf{Sig} \leftarrow\!\!\$\ \mathrm{sign}(\mathsf{sk}, m)$ takes as input private key sk and message m, and produces a signature Sig. Deterministic verification algorithm $b \leftarrow \mathsf{Vf}(\mathsf{pk}, m, \mathsf{Sig})$ takes as input public key pk, message m, and signature Sig, and outputs a Boolean b that signifies whether $\mathsf{Sig}$is a valid signature on m relative to public key pk. The standard definition of signature scheme security, existential unforgeability under chosen-message attack, has been introduced by Goldwasser, Micali, and Rivest [GMR88]. It states that the probability for an efficient adversary, given an oracle that generates valid signatures, to output a valid signature on a message that has not been queried to the oracle must be negligible. The security of a signature scheme can also be described by an ideal functionality $\mathcal{F}_{\mathrm{SIG}}$. We use the variant of the signature functionality $\mathcal{F}_{\mathrm{SIG}}$ that was introduced by Camenisch et al. [CDT19]. This version of the functionality is compatible with the modular NIZK proof technique introduced in the same paper.

4.4.3 Threshold Signature Schemes

A non-interactive threshold signature scheme TSIG consists of four algorithms $\mathrm{tkeygen}$, sign, $\mathrm{combine}$, and Vf. Threshold key generation $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{pk}) \leftarrow\!\!\$\ \mathrm{tkeygen}(\lambda, n, t)$ gets as input security parameter λ, total number of parties n, and threshold t. Each party can sign with their own secret key sk_i as above to generate a partial signature Sig_i. Any t valid signatures can be combined using $\mathrm{combine}$ into a full signature Sig, which is verified as in the non-threshold case. A signature produced honestly by any t parties verifies correctly, but any signature produced by less than t parties will not verify.

4.4.4 Public-Key Encryption

A public-key encryption scheme PKE consists of three algorithms $\mathrm{ekeygen}$, enc, and dec. Key-generation algorithm $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathrm{ekeygen}(\lambda)$ takes as input security parameter λ and outputs a pair of private key sk and public key pk. Probabilistic encryption algorithm $c \leftarrow\!\!\$\ \mathrm{enc}(\mathsf{pk}, m)$ takes as input message m and public key pk and produces ciphertext c. We also write $c \leftarrow \mathrm{enc}(\mathsf{pk}, m; r)$ where we want to emphasize that the encryption uses randomness r. Deterministic decryption $m \leftarrow \mathrm{dec}(\mathsf{sk}, c)$ takes as input ciphertext c and private key sk and recovers message m. Correctness requires that $\mathrm{dec}(\mathsf{sk}, \mathrm{enc}(\mathsf{pk}, m)) = m$ for all $(\mathsf{sk}, \mathsf{pk})$ generated by $\mathrm{ekeygen}$. For our work, we require semantic security as first defined by Goldwasser and Micali [GM84]. The scheme must additionally satisfy key privacy as defined by Bellare, Boldyreva, Desai, and Pointcheval [BBDP01], which states that, given a ciphertext c, it must be hard to determine the public key under which the ciphertext is encrypted.

4.4.5 Verifiable Random Functions

A verifiable random function VRF consists of three algorithms $\mathrm{vkeygen}$, eval, and check. Key generation $(vsk, vpk) \leftarrow\!\!\$\ \mathrm{vkeygen}(\lambda)$ takes as input the security parameter and outputs a pair of private key vsk and public key vpk. Deterministic evaluation $(y, \pi) \leftarrow \mathrm{eval}(vsk, x)$ takes as input secret key vsk and input value x, and produces as output the value y with proof π. Deterministic verification $b \leftarrow \mathrm{check}(vpk, x, y, \pi)$ takes as input public key vpk, input x, output y, and proof π, and outputs a Boolean that signifies whether the proof should be accepted.

The scheme satisfies *correctness* if honest proofs are always accepted. *Soundness* means that it is infeasible to produce a valid proof for a wrong statement. The scheme must satisfy *pseudo-randomness* which means that, given only vpk, the output y for a fresh input x is indistinguishable from a random output.

4.4.6 Non-Interactive Zero-Knowledge Proofs of Knowledge

Let $\mathcal{R}$ be a binary relation. For pairs $(x, w) \in \mathcal{R}$, x is called statement (i.e. public input) whereas w is called witness (i.e. private input). $\mathcal{L} = \{x, \exists\ w \text{ s.t. } (x, w) \in \mathcal{R}\}$ is called language of relation $\mathcal{R}$. A NIZK proof-of-knowledge system NIZK for language $\mathcal{L}$ comprises three algorithms: $\mathrm{zkcrsgen}$, prove and Vf. CRS generation $\mathrm{crs} \leftarrow\!\!\$\ \mathrm{zkcrsgen}(\lambda, \mathcal{R})$ takes as input the security parameter λ and a relation $\mathcal{R}$ and outputs a common reference string. On input of $(x, w) \in \mathcal{L}$ and crs, proof generation $\psi \leftarrow\!\!\$\ \mathrm{prove}(x, w, \mathrm{crs})$ returns a proof ψ. Proof ψ is verified by calling algorithm $b \leftarrow \mathrm{verify}(\psi, x, \mathrm{crs})$, which in turn outputs a Boolean that indicates whether the proof is valid or not.

Correctness for such a proof system means that honestly-generated proofs are always accepted. *Knowledge soundness* implies that a prover that produces a valid proof for some x must know a witness w with $(x, w) \in \mathcal{R}$, in the sense that w can be extracted. Finally, *zero-knowledge* ensures that the verification of correct statements yields nothing beyond the fact that they are correct. We describe the security of NIZK proofs of knowledge more formally using an ideal functionality $\mathcal{F}_{\text{NIZK}}$.

Functionality $\mathcal{F}^{R}_{\text{NIZK}}$

$\mathcal{F}_{\text{NIZK}}$ is parameterized by a relation R for which we can efficiently check membership. It keeps an initially empty list L of proven statements and a list L_0 of proofs that do not verify.

1. On input $(\texttt{prove}, y, w)$ from a party P, such that $(y, w) \in R$,[a] send $(\texttt{prove}, y)$ to $\mathcal{A}$.
2. Upon receiving a message $(\texttt{done}, \psi)$ from $\mathcal{A}$, with $\psi \in \{0,1\}^*$, record (y, ψ) in L and send $(\texttt{done}, \psi)$ to P.
3. Upon receiving $(\texttt{verify}, y, \psi)$ from some party P, check whether $(y, \psi) \in L$, then return 1 to P, or whether $(y, \psi) \in L_0$, then return 0 to P. If neither, then output $(\texttt{verify}, y, \psi)$ to $\mathcal{A}$and wait for receiving answer $(\texttt{witness}, w)$. Check $(y, w) \in R$ and if so, store (y, ψ) in L, else store it in L_0. If (y, ψ) is valid, then output 1 to P, else output 0.

[a] Inputs that do not satisfy the respective relation are ignored.

Figure 4.2: Non-interactive zero-knowledge functionality based on the one described by Groth et al. [GOS12a].

Our functionality $\mathcal{F}_{\text{NIZK}}$ is adapted from the work of Groth et al. [GOS12a], with a few modifications of which most are mainly stylistic. The most relevant difference is that we store a set L_0 of false statements that have been verified; we need this to ensure that a statement that was evaluated as false by one honest party will also be evaluated as false by all other honest parties. Otherwise $\mathcal{F}_{\text{NIZK}}$ has the two expected types of inputs `prove` and `verify`, and the adversary is allowed to delay proof generation unless $\mathcal{F}_{\text{NIZK}}$ is used in the context of responsive environments [CEK+16].

In the remainder of the chapter, we succinctly represent zero knowledge proofs of knowledge using the common notation introduced by Camenisch and Stadler [CS97], namely $\mathrm{PK}\,\{(x) : w\}$ denotes a proof of knowledge of *witness* w for *statement* x.

4.5 Security Formalization

4.5.1 Notation

We use sans-serif fonts to denote constants such as true or false, and `typewriter` fonts to denote string constants.

4.5.2 Universal Composition and MUC

In this section, we only recall basic notation and specific parts of the model that we need in this work. Details can be found in [Can01b, CDPW07, Can18].

The UC framework follows the simulation paradigm, and the entities taking part in the protocol execution (protocol machines, functionalities, adversary, and environment) are described as *interactive Turing machines* (ITMs). The execution is an interaction of *ITM instances* (ITIs) and is initiated by the environment $\mathcal{Z}$ that provides input to and obtains output from the protocol machines, and also communicates with adversary $\mathcal{A}$resp. simulator $\mathcal{S}$. The adversary has access to the protocols as well as functionalities used by them. Each ITI has an identity that consists of a party identifier pid and a session identifier sid. The environment and adversary have specific, constant identifiers, and ideal functionalities have party identifier $\perp$. The understanding here is that all ITIs that share the same code and the same sid are considered a *session* of a protocol. It is natural to use the same pid for all ITIs that are considered the same party.

ITIs can invoke other ITIs by sending them messages, new instances are created adaptively during the protocol execution when they are first invoked by another ITI. To use composition, some additional restrictions on protocols are necessary. In a protocol $\mu^{\phi\to\pi}$, which means that all calls within μ to protocol ϕ are replaced by calls to protocol π, both protocols ϕ and π must be *subroutine respecting*. This means, in a nutshell, that while those protocols may have further subroutines, all inputs to and outputs from subroutines of ϕ or π must only be given and obtained through ϕ or π, never by directly interacting with their subroutines. (This requirement is natural, since a higher-level protocol should never directly access the internal structure of ϕ or π; this would obviously hurt composition.) Also, protocol μ must be *compliant*. This roughly means that μ should not be invoking instances of π with the same sid as instances of ϕ, as otherwise these instances of π would interact with the ones obtained by the operation $\mu^{\phi\to\pi}$.

In summary, a protocol execution involves the following types of ITIs: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, instances of the protocol machines π, and (possibly) further ITIs invoked by $\mathcal{A}$or any instance of π (or their subroutines).

The contents of the environment's output tape after the execution is denoted by the random variable $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$, where $\lambda \in \mathbb{N}$ is the *security parameter* and $z \in \{0,1\}^*$ is the input to the environment $\mathcal{Z}$. The formal details of the execution are specified in [Can18]. We say that a protocol π UC-realizes a functionality $\mathcal{F}$ if

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z} : \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}},$$

where "$\approx$" denotes indistinguishability of the respective distribution ensembles, and ϕ is the dummy protocol that simply relays all inputs to and outputs from the functionality $\mathcal{F}$.

Multi-protocol UC. The standard UC framework does not allow to modularly prove protocols in which, e.g., a zero-knowledge proof system is used to prove that a party has performed a certain evaluation of a cryptographic scheme correctly. Camenisch, Drijvers, and Tackmann [CDT19] recently showed how this can be overcome. In a nutshell, they start from the standard $\mathcal{F}_{\text{NIZK}}^{R}$-functionality which is parameterized by a relation R, and show that if R is described in terms of evaluating a protocol, then the protocol can equivalently be evaluated outside of the functionality, and even used to realize another functionality $\mathcal{F}$. This results in a setting where $\mathcal{F}_{\text{NIZK}}$ validates a pair (y, w) of statement y and witness w by "calling out" to the other functionality $\mathcal{F}$. We use this proof technique extensively in this work.

4.5.3 The Privacy-Preserving Token Functionality

The functionality $\mathcal{F}_{\text{TOKEN}}$ realized by our privacy-preserving token system is formalized in Figure 4.3. To keep the presentation simple, the functionality formalizes the guarantees for the case of a single token issuer I. The functionality initially requires registration authority A, certifier C, and issuer I to initialize. (This corresponds to the fact that all protocol steps depend on those parties' keys.) Likewise, regular parties P have to generate and register their keys before they can perform operations. Each party can then `read` the tokens they own and generate `transfer` transactions that reference those tokens and transfers them to one or more receivers. Issuers can additionally `issue` new tokens. In the inputs and outputs of the functionality, v always represents the value of a token, and cm serves as a handle identifying the token (it stems from the commitment that represents the token on the ledger). Finally, the functionality specifies which information is potentially leaked to the adversary, and which operations the adversary can perform in the name of corrupted parties.

4.5.4 Set-Up Functionalities

Our protocol requires a number of set-up functionalities to be available.

Common reference string. Functionality $\mathcal{F}_{\text{CRS}}$ (see Figure 4.5) provides a string that is sampled at random from a given distribution and accessible to all participants. All parties can simply query $\mathcal{F}_{\text{CRS}}$ for the reference string. The functionality is generally used to generate common public parameters used in a cryptographic scheme. Functionality $\mathcal{F}_{\text{CRS}}$ is parameterized by a CRS generator crsgen, which on input security parameter λsamples a fresh string $crs \leftarrow\!\!\$\ \text{crsgen}(\lambda)$.

Transaction ledger. We describe a simplified transaction ledger functionality as $\mathcal{F}_{\text{LEDGER}}$ in Figure 4.5. In a nutshell, every party can append bit strings to a globally available ledger, and every party can retrieve the current ledger.

The functionality intentionally idealizes the guarantees achieved by a real-world ledger; transactions are immediately appended, final, and available to all parties. We also use $\mathcal{F}_{\text{LEDGER}}$ as a local functionality. These simplifications are intended to keep the paper more digestible.

Secure and private message transfer. Functionality $\mathcal{F}_{\text{SMT}}$ provides a message transfer mechanism between parties. The functionality builds on the ones described by Canetti and Krawczyk [CK02], but additionally hides the sender and receiver of a message, if both are honest. This is required since our protocol passes information between transacting parties, and leaking the communication pattern to the adversary would revoke the anonymity otherwise provided by our protocol.

More formally, functionality $\mathcal{F}_{\text{SMT}}$ models a secure channel between a sender S and a receiver R. In comparison to the functionality introduced by Canetti and Krawczyk [CK02], however, our functionality additionally provides privacy and hides the parties that are involved in the transmission.

Public-key registration. The registration functionality $\mathcal{F}_{\text{REG}}$ models a public-key infrastructure. It allows each party P to input one value $x \in \{0,1\}^*$ and makes the pair (P, x) available to all other parties. This is generally used to publish public keys, binding them to the identity of a party.

Anonymous authentication. As our protocol is in the permissioned setting but supposed to provide privacy, we need anonymous credentials to authorize transactions. Our schemes integrate well with the Identity Mixer family of protocols [CH02]. Yet, as these topics are not the core interest of this paper, we abstract the necessary mechanisms in the functionality $\mathcal{F}_{\text{A-AUTH}}$ as depicted in Figure 4.8.

In a nutshell, the functionality allows parties to first register and then "authorize" commitments; the functionality returns "proofs" ψ assuring that the party's identity is contained in a certain position of that commitment.

Privacy-preserving token functionality $\mathcal{F}_{\text{TOKEN}}$

Functionality $\mathcal{F}_{\text{TOKEN}}$ stores a list of registered users and an initially empty map Records. The session identifier is of the form $sid = (A, C, I, sid')$.

- Upon input `init` from $P \in \{A, C, I\}$, output to $\mathcal{A}(\texttt{initialized}, P)$. (This must happen for all three before anything else.)
- Upon input `register` from a party P, if P is unregistered, then mark P as registered and output $(\texttt{registered}, P)$ to $\mathcal{A}$. (Otherwise ignore.)
- Upon input `read` from a registered party P, issue $(\texttt{read?}, P)$ to $\mathcal{A}$. Upon receiving response $(\texttt{read!}, P)$ from $\mathcal{A}$, return to P a list of all records of the type (cm, v) that belong to P.
- Upon input $(\texttt{issue}, v)$ from I, output $(\texttt{issue}, v)$ to $\mathcal{A}$. Receiving from $\mathcal{A}$a response $(\texttt{issue}, cm)$, if $\mathsf{Records}[cm] \neq \perp$ then abort, else set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{alive})$. Return $(\texttt{issued}, cm)$ to I.
- Upon receiving an input $(\texttt{issue}, v, cm)$ from $\mathcal{A}$, where I is corrupt, check and record the commitment as in the previous step. Return to $\mathcal{A}$.
- Upon input $\left(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$ from an honest party P, where P and all R_j for $j = 1, \ldots, n$ are registered, proceed as follows.
  1. If, for any $i \in \{1, \ldots, m\}$, $\mathsf{Records}[cm_i] = \perp$ then abort, else set $(v_i^{\text{in}}, P_i', st_i) \leftarrow \mathsf{Records}[cm_i]$.
  2. If, for any $i \in \{1, \ldots, m\}$, $st_i \neq \texttt{alive}$ or $P_i' \neq P$, then abort.
  3. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^n v_j^{\text{out}}$ then abort.
  4. Let L be an empty list. For all $j = 1, \ldots, n$, if R_j is corrupt then append to L the information $(j, P, R_j, v_j^{\text{out}})$. Output $(\texttt{transfer}, m, n, L)$ to $\mathcal{A}$.
  5. Receiving from $\mathcal{A}$a response $(\texttt{transfer}, (cm_j^{\text{out}})_{j=1}^n)$, if $\mathsf{Records}[cm_j^{\text{out}}] \neq \perp$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\mathsf{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{delayed})$ for all $j \in \{1, \ldots, n\}$ and set $\mathsf{Records}[cm_i] \leftarrow (v_i^{\text{in}}, P', \texttt{consumed})$ for all $i \in \{1, \ldots, m\}$.
  6. Return $(\texttt{transferred}, (cm_j^{\text{out}})_{j=1}^n)$ to P.
- On $(\texttt{transfer}, P, (cm_i^{\text{in}})_{i=1}^m, (R_j, v_j^{\text{out}}, cm_j^{\text{out}})_{j=1}^n)$ where P is corrupt, proceed analogously to above except for using the given output commitments.
- Upon receiving an input $(\texttt{deliver}, cm)$ from $\mathcal{A}$with $\mathsf{Records}[cm] = (v, P, \texttt{delayed})$ for some v, set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{alive})$. If C is corrupted, then output P to $\mathcal{A}$.

Figure 4.3: Privacy-preserving token functionality.

Functionality $\mathcal{F}_{\text{CRS}}^{\text{crsgen}}$

$\mathcal{F}_{\text{CRS}}$ is parameterized by a probabilistic algorithm crsgen. Initially, it sets $crs \leftarrow\!\!\$\ \text{crsgen}(\lambda)$.

1. On input `read` from a party P, return crs to P.

Figure 4.4: Common reference string.

Ledger functionality $\mathcal{F}_{\text{LEDGER}}$

Functionality $\mathcal{F}_{\text{LEDGER}}$ stores an initially empty list L of bit strings.

- Upon input $(\texttt{append}, x)$ from a party P, append x to L. If P is corrupt then send $(\texttt{append}, x, P)$ to $\mathcal{A}$, else return to P.
- Upon input `retrieve` from a party P or $\mathcal{A}$, return L.

Figure 4.5: Ledger functionality.

The functionality allows to further bind the proof to a bit string m, which intuitively can be understood as "party P (as referenced in the commitment) signs message m." The exact reason for this mechanism will become clear in the protocol description in Section 4.6.4.

Our description of $\mathcal{F}_{\text{A-AUTH}}$ is simplistic and tailored to an easy treatment in our proofs. For a complete composable model of anonymous authentication schemes, see e.g. the work of Camenisch, Dubovitskaya, Haralambiev, and Kohlweiss [CDHK15a].

Digital signatures. We use the variant of the signature functionality $\mathcal{F}_{\text{SIG}}$ that was introduced by Camenisch et al. [CDT19]. This version of the functionality is compatible with the modular NIZK proof technique introduced in the same paper.

Distributed key generation. Functionality $\mathcal{F}_{\text{DKG}}$ idealizes a distributed key-generation protocol such as, for discrete-log based schemes, the one of Gennaro et al. [GJKR07]. The simplified functionality given in Figure 4.10 is not directly realizable since it does not model that, e.g., the communication may be delayed or prevented by the adversary. We decided to still use this version to simplify the overall treatment.

Secure message transmission functionality $\mathcal{F}_{\text{SMT}}$

Functionality $\mathcal{F}_{\text{SMT}}$ is for transmitting messages in a secure and *private* manner.

- Upon input $(\texttt{send}, R, m)$ from a party S:
  - If both S and R are honest, provide a private delayed output $(\texttt{sent}, S, R, m)$ to R.
  - If at least one of S and R is corrupt, provide a public delayed output $(\texttt{sent}, S, R, m)$ to $\mathcal{A}$'s queue.

Figure 4.6: Secure message transmission functionality.

Registration functionality $\mathcal{F}_{\text{REG}}$

Functionality $\mathcal{F}_{\text{REG}}$ is for registering users' public keys. It stores an initially empty list of users $\{U\}$ and an empty list $\{X\}$ of pairs of users and public keys.

- Upon input $(\texttt{register}, x)$ from a party P where $P \notin \{U\}$ and $(\star, x) \notin \{X\}$, set $\{U\} \leftarrow \{U\} \cup \{P\}$ and $\{X\} \leftarrow \{X\} \cup \{(P, x)\}$ and output $(\texttt{registered}, (P, x))$ to $\mathcal{A}$.
- Upon input $(\texttt{lookup}, P')$ from a party P, check if $P' \in \{U\}$. If not, return $\perp$, else return (P', x').

Figure 4.7: Registration functionality.

Extended anonymous authentication functionality $\mathcal{F}_{\text{A-AUTH}}$

Functionality $\mathcal{F}_{\text{A-AUTH}}$ is parameterized by a commitment opening algorithm open. It stores an initially empty $\{U\}$ of registered users, and an initially empty list of records.

- Upon input $\texttt{register}$ from a party P where $P \notin \{U\}$, set $\{U\} \leftarrow \{U\} \cup \{P\}$ and output $(\texttt{registered}, P)$ to $\mathcal{A}$.
- Upon input $(\texttt{lookup}, P')$ from a party P, return (the result of) $P' \in \{U\}$.
- Upon input $(\texttt{prove}, crs, cm, r_{\text{cm}}, x, y, m)$ from party P, if $\text{open}(crs, cm, (x, P, y), r_{\text{cm}})$ then generate a proof ψ, store the record $(\texttt{transfer}, crs, cm, \psi, m)$ internally and output ψ to P.
- Upon input $(\texttt{verify}, crs, cm, \psi, m)$ from some P, look up if there is a record $(\texttt{transfer}, crs, cm, \psi, m)$ and output success (only) if it exists.

Figure 4.8: Extended anonymous registration functionality.

Functionality $\mathcal{F}_{\text{SIG}}^{(\text{skeygen},\text{sign},\mathsf{Vf})}$

Functionality $\mathcal{F}_{\text{SIG}}$ (see Figure 4.9) requires that $sid = (S, sid')$, where S is the party identifier of the sender. Set $\{C\}$, initially empty, specifies the set of currently corrupted parties. The functionality keeps a set $\{S\}$of properly signed messages.

0. Upon the first activation from S, run $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \text{skeygen}(\lambda)$, where $\lambda$is obtained from the security parameter tape, and store $(\mathsf{sk}, \mathsf{pk})$.
1. Upon input $\texttt{pubkey}$ from party S, output $(\texttt{pubkey}, \mathsf{pk})$ to S.
2. Upon input $(\texttt{sign}, m)$ from party S with $m \in \{0,1\}^*$, compute $\mathsf{Sig} \leftarrow\!\!\$\ \text{sign}(\mathsf{sk}, m)$. Set $\{S\} \leftarrow \{S\} \cup \{m\}$ and output s to S.
3. Upon input $(\texttt{verify}, \mathsf{pk}', m', s')$ from party P, compute $b \leftarrow \mathsf{Vf}(\mathsf{pk}', m', s')$. If $S \notin \{C\} \wedge \mathsf{pk} = \mathsf{pk}' \wedge b \wedge m' \notin \{S\}$ then output $(\texttt{result}, 0)$ to P. Else output $(\texttt{result}, b)$ to P.
4. Upon input $(\texttt{corrupt}, P)$ from the adversary, set $\{C\} \leftarrow \{C\} \cup \{P\}$. If $P = S$, then additionally output $\mathsf{sk}$to $\mathcal{A}$.

Figure 4.9: Signature functionality.

Functionality $\mathcal{F}_{\text{DKG}}$

$\mathcal{F}_{\text{DKG}}$ is parameterized by a PPT algorithm tkeygen. The session identifier sid specifies the total number of certifiers n and the threshold bound t.

- Upon input $\texttt{init}$ from a party C_i:
  - If no keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk})$ are stored yet, generate $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk}) \leftarrow\!\!\$\ \text{tkeygen}(\lambda, n, t)$.
  - Return $(\mathsf{sk}_i, \mathsf{pk})$ to C_i.

Figure 4.10: Distributed key generation functionality.

4.6 Privacy-Preserving Auditable UTXO

This section describes the complete protocol. We begin by introducing the core ideas and mechanisms. Section 4.6.4 then describes multi-input multi-output transactions, followed by Section 4.6.5 that assembles all pieces and describes the full protocol. Section 4.6.6 introduces the extension that makes the protocol auditable.

4.6.1 Core Protocol Ideas

The protocol represents tokens as commitments $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, P))$ that are stored on $\mathcal{F}_{\text{LEDGER}}$, where v is the value and P is the current owner. Issuers can create new tokens in their own name. Transferring tokens (v, P) to a party R means replacing the commitment to (v, P) with a commitment to (v, R). We now describe the protocol steps in more detail, but still at an informal level.

To issue a token of value v, the issuer generates a new commitment $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, I))$, which means a token with value v is created with owner I. The protocol generates a proof

$$\psi_0 \leftarrow \text{PK}\left\{(r_{\text{cm}}) : \text{open}(crs, cm, r_{\text{cm}}, (v, I)) = \mathsf{true}\right\},$$

which shows that the commitment contains the expected information. Issuer I also creates a signature Sig on message (v, cm, ψ_0). The information written to $\mathcal{F}_{\text{LEDGER}}$ is $tx = (\texttt{issue}, v, cm, \psi_0, \mathsf{Sig})$.

A party can transfer a token identified by a commitment cm to a receiver R by generating a new commitment $(cm', r'_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, R))$. She generates a first NIZK ψ_1 showing that cm' contains the correct information and that the receiver is registered, and a second proof ψ_2 of eligibility (i.e. the initiator of the transfer owns cm) using $\mathcal{F}_{\text{A-AUTH}}$. The information written to $\mathcal{F}_{\text{LEDGER}}$ is $tx = (\texttt{transfer}, cm', \psi_1, \psi_2)$. At this point, we cannot yet describe how P proves that (a) cm is a valid commitment on the ledger—we cannot include cm in the transaction as that would hurt privacy—and (b) that P is not double-spending cm. These aspects will be covered in the next steps. Party P also sends the message $(\texttt{token}, cm', r'_{\text{cm}}, v)$ to R *privately*.

So far, we have shown how to transfer a single token from a party P to a receiver R. Following sections show how to (i) make sure that only *valid and unspent tokens* are transferred; and (ii) support multi-input multi-output transfers.

4.6.2 Certification via Blind Signatures

(Threshold) blind signature functionality $\mathcal{F}_{\text{BLINDSIG}}^{(\text{TSIG,commit})}$

Functionality $\mathcal{F}_{\text{BLINDSIG}}$ requires that $sid = ((C_1, \ldots, C_n), t, \ell, sid')$, where $C_1, \ldots, C_n$ are the party identifiers of the signers. It is parameterized by the (deterministic) commit algorithm of the commitment as well as the a threshold signature scheme $\text{TSIG} = (\text{tkeygen}, \text{sign}, \mathsf{Vf})$. The functionality keeps an initially empty set $\{S\}$of signed messages.

- Upon `init` from some C_i, run $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{pk}) \leftarrow\!\!\$\ \text{tkeygen}(\lambda, n, t, \ell)$, where $\lambda$is obtained from the security parameter tape, and store $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk})$. Output $(\texttt{init}, C_i)$ to $\mathcal{A}$.
- Upon input `pubkey` from party P, return $(\texttt{pubkey}, \mathsf{pk})$.
- On input $(\texttt{request}, crs, r_{\text{cm}}, (m_1, \ldots, m_\ell))$ from party P:
  1. Compute $cm \leftarrow \text{commit}(crs, (m_1, \ldots, m_\ell); r_{\text{cm}})$ and store it internally along with the messages and randomness.
  2. Send delayed output $(\texttt{request}, P, crs, cm)$ to each C_i, $i \in \{1, \ldots, n\}$.
- Upon input $(\texttt{sign}, cm)$ from C_i:
  1. If no record with commitment cm exists, then abort.
  2. If there is a record $((m_1, \ldots, m_\ell), S) \in \{S\}$ with $S \subseteq \{C_1, \ldots, C_n\}$, update the record with $S \leftarrow S \cup \{C_i\}$. Else set $\{S\} \leftarrow \{S\} \cup \{((m_1, \ldots, m_\ell), \{C_i\})\}$.
  3. If $|S| \geq t$, then compute $\mathsf{Sig} \leftarrow \text{sign}(\mathsf{sk}, (m_1, \ldots, m_\ell))$ and output Sig to requestor P.
- Upon input $(\texttt{verify}, \mathsf{pk}', (m_1, \ldots, m_\ell), \mathsf{Sig})$ from P, compute $b \leftarrow \mathsf{Vf}(\mathsf{pk}', (m_1, \ldots, m_\ell), \mathsf{Sig})$. If $\mathsf{pk} = \mathsf{pk}' \wedge b \wedge \big(((m_1, \ldots, m_\ell), S) \notin \{S\} \vee |S| < t\big)$ then output $(\texttt{result}, \mathsf{false})$ to P. Else output $(\texttt{result}, b)$ to P.
- Upon input $(\texttt{seckey}, i)$ from $\mathcal{A}$, if C_i is corrupted, then return sk_i.

Figure 4.11: Blind signature functionality, threshold version.

The problem of verification of token validity during transfer is resolved by *certification*. We consider a specific party, called a *certifier* C, which vouches for the validity of the token (v, P) stored as a commitment cm

on $\mathcal{F}_{\text{LEDGER}}$ by issuing a signature Sigon (v, P). In the proof ψ_1, P refers to signature Siginstead of commitment cm.

A naive implementation of the above scheme would require party P to reveal the content (v, P) of cm to certifier C, so that the latter issues the corresponding signature Sig. Namely prior to signing, C checks that cm opens to (v, P) and that cm is stored on $\mathcal{F}_{\text{LEDGER}}$. Disclosing the pair (v, P) to C is both undesired and unnecessary. Instead we rely on a *blind signature* protocol, in which C learns only the commitment cm, but not its contents, and *blindly* signs the contents.

While C learns cm during the protocol, it will not be able to leverage the data on $\mathcal{F}_{\text{LEDGER}}$ to trace when P makes use of the corresponding signature Sig. More precisely, within ψ_1, party P only proves knowledge of signature $\mathsf{Sig}$and does not reveal it.

Note that a malicious certifier can essentially generate tokens by providing its signature without checking for existence of the commitment on $\mathcal{F}_{\text{LEDGER}}$. Therefore, in Section 4.7.7, we describe how the certification task can be distributed, so that no single party has to be trusted for verification. Figure 4.11 shows a threshold blind signature ideal functionality, which we will use in the description of our solution in Section 4.6.5.

4.6.3 Serial Numbers Prevent Double-Spending

Double-spending prevention is achieved via a scheme that is inspired by Zerocash [BSCG$^+$14] in that it uses a VRF to compute serial numbers for tokens when they are spent. The VRF key is here, however, bound to a user identity via a signature from the registration authority. On a very high level, the above protocol is extended as follows.

1. Each user P creates a VRF key pair (vsk, vpk). They obtain a signature Sig_A from registration authority A that binds vsk to their identity P.
2. Each commitment contains an additional value ρ.
3. During transfer, the value ρ is used to derive the serial number $(sn, \pi) \leftarrow \text{eval}(vsk, \rho)$. The transaction stored in $\mathcal{F}_{\text{LEDGER}}$ also contains sn.
4. We cannot store the VRF proof π on $\mathcal{F}_{\text{LEDGER}}$, as it is bound to vpk and would deanonymize P. Therefore, P proves knowledge of signature Sig_A, which binds vpk to her identity, and proves that $\text{check}(vpk, \rho, sn, \pi) =$ true through a NIZK proof.

It is important to note that authority A must be trusted for preventing double-spending, since it could easily register two different VRF keys for the same user. It is therefore recommended to implement A in a distributed fashion.

The proof ψ_1 made by P during a transfer of the token (v, P, ρ^{in}) is then

$$\psi_1 \leftarrow \text{PK}\big\{\big(r'_{\text{cm}}, \mathsf{Sig}_A, \mathsf{Sig}_C, R, P, \rho^{\text{in}}, \rho^{\text{out}}, \pi, v\big) : \mathsf{Vf}(\mathsf{pk}_C, (v, P, \rho^{\text{in}}), \mathsf{Sig}_C) \wedge \text{open}(crs, cm', (v, R, \rho^{\text{out}}), r'_{\text{cm}}) \wedge \mathsf{Vf}(\mathsf{pk}_A, (P, vpk), \mathsf{Sig}_A) \wedge \text{check}(vpk, \rho^{\text{in}}, sn, \pi)\big\},$$

which can be parsed as follows: prior to the transfer, P obtains signature Sig_C on (v, P, ρ^{in}) under pk_C from C. The first condition in the proof statement checks that P knows signature Sig_C on the triplet (v, P, ρ^{in}). The second condition checks that the new commitment cm' contains the same value v. These two conditions, together with trust in the correctness of C, ensure that the token corresponding to cm' is properly derived from a token existing on $\mathcal{F}_{\text{LEDGER}}$. The third condition checks that the VRF public key vpk indeed belongs to P, and the fourth condition checks that the computation of the serial number sn is correct. These two conditions, together with trust in the correctness of A, prevent token (v, P, ρ^{in}) from being double-spent.

4.6.4 Multi-Input Multi-Output Transactions

Multi-input multi-output transactions allow a sender to transfer tokens contained in multiple commitments at once, and to split the accumulated value into multiple outputs for potentially different receivers. We therefore modify the transaction format to contain multiple inputs and multiple outputs. We also have to extend the NIZK: besides the fact that we have to prove consistency of multiple inputs and multiple outputs, we now have to show that the *sum* of the inputs equals the *sum* of the outputs.

Due to arithmetics in finite algebraic structures, we also have to prove that no wrap-arounds occur. This is achieved, as in previous work, by the use of range proofs. For a given value $\mathsf{max} \in \{1, \ldots, p\}$, the condition is that $0 \leq v \leq \mathsf{max}$ for any value v that appears in an output commitment.

The proof, in more detail, now becomes

$$\begin{aligned}\psi_1 \leftarrow \mathrm{PK}\big\{\big((\mathsf{Sig}_i, v_i^{\mathrm{in}}, \rho_i^{\mathrm{in}}, \pi_i)_{i=1}^m, P, \mathsf{Sig}_A, (R_j, r_{\mathrm{cm}}^j, v_j^{\mathrm{out}}, \rho_j^{\mathrm{out}}, vpk_j, \mathsf{Sig}_A^j)_{j=1}^n\big) :\\ \forall i \in \{1,\ldots,m\} : \mathsf{Vf}(\mathsf{pk}_C, (v_i^{\mathrm{in}}, P, \rho_i^{\mathrm{in}}), \mathsf{Sig}_i)\\ \wedge \forall j \in \{1,\ldots,n\} : \mathrm{open}(crs, cm_j, (v_j^{\mathrm{out}}, R_j, \rho_j^{\mathrm{out}}), r_{\mathrm{cm}}^j)\\ \wedge \mathsf{Vf}(\mathsf{pk}_A, (P, vpk), \mathsf{Sig}_A)\\ \wedge \forall j \in \{1,\ldots,n\} : \mathsf{Vf}(\mathsf{pk}_A, (R_j, vpk_j), \mathsf{Sig}_A^j)\\ \wedge \forall i \in \{1,\ldots,m\} : \mathrm{check}(vpk, \rho_i^{\mathrm{in}}, sn_i, \pi_i)\\ \wedge \sum_{i=1}^m v_i^{\mathrm{in}} = \sum_{j=1}^m v_j^{\mathrm{out}} \wedge \forall j \in \{1,\ldots,n\} : 0 \leq v_j^{\mathrm{out}} \leq \mathsf{max}\big\}. \quad (4.1)\end{aligned}$$

The processing of the transaction is analogously modified to check this more complex NIZK. We now argue that the statement proved in the NIZK indeed guarantees the consistency of the system.

The first sub-statement (together with the honesty of C) guarantees that all commitments used as inputs indeed exist in the ledger, and the fact that the commitment is binding further implies that the values $(v_i^{\mathrm{in}}, P, \rho_i^{\mathrm{in}})$ indeed correspond to the expected state of the system. The next sub-statement shows that the output commitments indeed contain the expected values $(v_j^{\mathrm{out}}, R_j, \rho_j^{\mathrm{out}})$. The subsequent two statements ensure that all parties are properly registered on the system, and the statement $\mathrm{check}(vpk, \rho_i^{\mathrm{in}}, sn_i, \pi_i)$ prevents double-spending by showing that the serial numbers are computed correctly.

The final two equations guarantee the global consistency of the system: the summation equation then shows that no tokens have been created or destroyed in this transaction. Finally, the range proof shows that all outputs contain a value in the valid range, which avoids wrap-arounds.

4.6.5 The Protocol

This section describes the protocol sketched in the above sections more formally. The protocol has a bit $\mathit{registered} \leftarrow \mathsf{false}$ and keeps an initially empty list of commitments. We begin by describing the protocol for a regular user P of the system.

- Upon input `register`, if *registered* is set, then return. Else, retrieve the public keys of A and C from $\mathcal{F}_{\text{REG}}$. Query crs from $\mathcal{F}_{\text{CRS}}$. Generate a VRF key pair (vsk, vpk) and send a message $(\texttt{register}, vpk)$ to A via $\mathcal{F}_{\text{SMT}}$ to obtain a signature Sig_A on (P, vpk). If all steps succeeded, then set $\mathit{registered} \leftarrow \mathsf{true}$ send `register` to $\mathcal{F}_{\text{A-AUTH}}$.
- Process pending messages and retrieve new data from $\mathcal{F}_{\text{LEDGER}}$. This is a subroutine called from functions below.
  - For transactions $tx = (\texttt{issue}, v, cm, \psi_0, \mathsf{Sig})$ from $\mathcal{F}_{\text{LEDGER}}$, validate ψ_0 by inputting $(\texttt{verify}, (crs, cm, v, I), \psi_0)$ to $\mathcal{F}_{\text{NIZK}}$ and verify Sig via $\mathsf{Vf}(\mathsf{pk}_I, (v, cm, \psi_0), \mathsf{Sig})$. If both checks succeed, record cm as a valid commitment.

  - For transactions $tx = (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$, check the serial numbers $sn_1, \ldots, sn_m$ for uniqueness, validate ψ_1 via $\mathcal{F}_{\text{NIZK}}$ and verify $\psi_{2,1}, \ldots, \psi_{2,n}$ via $(\texttt{verify}, crs, cm_i, \psi_{2,i}, \mathbf{m})$ to $\mathcal{F}_{\text{A-AUTH}}$ for $\mathbf{m} = ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$. If all checks succeed, then store $cm_1, \ldots, cm_m$ as valid.
  - For each incoming message $(\texttt{sent}, S, P, m)$ buffered from $\mathcal{F}_{\text{SMT}}$, parse m as $(\texttt{token}, cm, r_{\text{cm}}, v, \rho)$ and test whether the commitment is correct, i.e. whether it holds that $\text{open}(crs, cm, r_{\text{cm}}, (v, P, \rho)) = \text{true}$. Check whether there is a `transfer` transaction tx that appears in $\mathcal{F}_{\text{LEDGER}}$ and contains cm. If all checks are successful, then input $(\texttt{request}, r_{\text{cm}}, (v^{\text{in}}, P, \rho^{\text{in}}))$ to $\mathcal{F}_{\text{BLINDSIG}}$ and wait for a response Sig_{cm}. Store the complete information in the internal list.

- Upon input `read`, if $\neg$*registered* then abort, else first process pending messages. Then return a list of all unspent assets (cm, v) owned by the party.

- Upon input $(\texttt{issue}, v)$, assuming that *registered*, process pending messages and proceed as follows.

  1. Choose a uniformly random ρ and create a commitment $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, I, \rho))$.
  2. Compute a proof

  $$\psi_0 \leftarrow \text{PK}\left\{(r_{\text{cm}}, \rho) : \text{open}(crs, cm, r_{\text{cm}}, (v, I, \rho)) = \mathsf{true}\right\},$$

  where I and v are publicly known; this is achieved by sending $(\texttt{prove}, x, w)$ to $\mathcal{F}_{\text{NIZK}}$, where the statement is $x = (crs, cm, v, P)$ and the witness is $w = (r_{\text{cm}}, \rho)$. Compute a signature $\mathsf{Sig} \leftarrow\!\!\$\ \text{sign}(\mathsf{sk}_I, (v, cm, \psi_0))$.
  3. Send to $\mathcal{F}_{\text{LEDGER}}$ the input $(\texttt{append}, (\texttt{issue}, v, cm, \psi_0, \mathsf{Sig}))$.
  4. Store tuple $(cm, r_{\text{cm}}, v, \rho)$ internally and return $(\texttt{issued}, cm)$.

- Upon input $\left(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$, assuming that *registered*, query $(\texttt{lookup}, R_j)$ to $\mathcal{F}_{\text{REG}}$ for all $j = 1, \ldots, n$ in order to make sure that R_j is registered. Then process pending messages and proceed as follows.

  1. If, for any $i \in \{1, \ldots, m\}$, there is no recorded commitment $(cm_i, r_{\text{cm}}^i, v_i^{\text{in}}, P, \rho_i^{\text{in}})$, then abort.
  2. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^m v_j^{\text{out}}$ then abort.
  3. Choose uniformly random ρ_j^{out} for $j = 1, \ldots, n$, and create commitments $(cm_j, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (v_j^{\text{out}}, R_j, \rho_j^{\text{out}}))$.
  4. Compute the serial numbers as $(sn_i, \pi_i) \leftarrow \text{eval}(vsk, \rho_i^{\text{in}})$, for $i = 1, \ldots, m$.
  5. Compute proof ψ_1 as in Equation (4.1).
  6. Set $\mathbf{m} \leftarrow ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$ For each $i = 1, \ldots, m$, send $(\texttt{prove}, cm_i^{\text{in}}, r_i, v_i^{\text{in}}, \rho_i^{\text{in}}, \mathbf{m})$ to obtain $\psi_{2,i}$.
  7. Send $(\texttt{token}, cm_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ to R_j via $\mathcal{F}_{\text{SMT}}$, for each $1 \leq j \leq m$, and send to $\mathcal{F}_{\text{LEDGER}}$ the input
  $$(\texttt{append}, (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)).$$
  8. Delete cm_i^{in} from the internal state and return $(\texttt{transferred}, (cm_j)_{j=1}^n)$.

- Upon receiving $(\texttt{sent}, S, P, m)$ from $\mathcal{F}_{\text{SMT}}$, buffer it for later processing. Respond `ok` to sender S.

The protocol machines for parties C and A are easier to describe. Certifier C checks the validity of a commitment and signs if it finds the commitment in the ledger. In more detail:

1. Upon input `init` obtain crs from $\mathcal{F}_{\text{CRS}}$ and input `init` to $\mathcal{F}_{\text{BLINDSIG}}$.

2. Upon receiving $(\texttt{request}, P, crs', cm)$ from $\mathcal{F}_{\text{BLINDSIG}}$, check that $crs = crs'$. Query $\mathcal{F}_{\text{LEDGER}}$ for the entire ledger. For each yet unprocessed transaction tx on $\mathcal{F}_{\text{LEDGER}}$, validate the proofs as described in the party protocol. Check whether cm is marked as a valid commitment.
3. If the above check is successful, send $(\texttt{sign}, cm)$ to $\mathcal{F}_{\text{BLINDSIG}}$.

Certification authority A signs VRF public keys of parties.

1. Upon $\texttt{init}$, generate a key pair $(\mathsf{sk}_A, \mathsf{pk}_A) \leftarrow\!\!\$\ \text{skeygen}(\lambda)$ for the signature scheme and input $(\texttt{register}, \mathsf{pk}_A)$ to $\mathcal{F}_{\text{REG}}$.
2. When activated, input $\texttt{retrieve}$ to $\mathcal{F}_{\text{SMT}}$ to obtain the next message. Let it be m from P. If no message has been signed for P yet, then sign $\mathsf{Sig}_A \leftarrow \text{sign}(\mathsf{sk}_A, (P, m))$ and send Sig_A via $\mathcal{F}_{\text{SMT}}$ back to P.

4.6.6 Auditing

The auditing capability we implement associates to each user U an auditor AU. Auditor AU has the capabilities to decrypt all transaction information associated to U, such as the transaction outputs that are associated with U, as well as the full transactions issued by U. The set of auditors is denoted by $\mathbf{AU}$.

We formalize the guarantees in a functionality $\mathcal{F}_{\text{ATOKEN}}$ described in the following. Functionality $\mathcal{F}_{\text{ATOKEN}}$ stores a list of registered users and an initially empty map $\mathsf{Records}$. The session identifier is of the form $sid = (A, C, I, \mathbf{AU}, sid')$.

- Upon input $\texttt{init}$ from $P \in \{A, C\} \cup \mathbf{AU}$, output to $\mathcal{A}(\texttt{initialized}, P)$. (This must happen for all before anything else.)
- Inputs $\texttt{register}$, $\texttt{read}$, and $\texttt{issue}$ are treated as in $\mathcal{F}_{\text{TOKEN}}$.
- Upon input $(\texttt{bind}, U, AU)$ from A, where U is a registered user and $AU \in \mathbf{AU}$ is an initialized auditor, and there is not yet a pair (U, AU') with $AU \neq AU' \in \mathbf{AU}$, record (U, AU) and output $(\texttt{bound}, U, AU)$ to $\mathcal{A}$.
- Upon input $\left(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$ from an honest party P, where P and all R_j for $j = 1, \ldots, n$ are registered, proceed as follows.
  1. If, for any $i \in \{1, \ldots, m\}$, $\mathsf{Records}[cm_i] = \bot$ then abort, else set $(v_i^{\text{in}}, P_i', st_i) \leftarrow \mathsf{Records}[cm_i]$.
  2. If, for any $i \in \{1, \ldots, m\}$, $st_i \neq \texttt{alive}$ or $P_i' \neq P$, then abort.
  3. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^n v_j^{\text{out}}$ then abort.
  4. Let L be an empty list. For all $j = 1, \ldots, n$, if R_j or its auditor AU_j are corrupt, then append to L the information $(P, R_j, v_j^{\text{out}})$. If the auditor AU of P is corrupt, include the information for all inputs and all outputs. Output $(\texttt{transfer}, L)$ to $\mathcal{A}$.
  5. Receiving from $\mathcal{A}$a response $(\texttt{transfer}, (cm_j^{\text{out}})_{j=1}^n)$, if $\mathsf{Records}[cm_j^{\text{out}}] \neq \bot$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\mathsf{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{alive})$ for all $j \in \{1, \ldots, n\}$ and set $\mathsf{Records}[cm_i] \leftarrow (v_i^{\text{in}}, P, \texttt{consumed})$ for all $i \in \{1, \ldots, m\}$.
  6. Return $(\texttt{transferred}, (cm_j^{\text{out}})_{j=1}^n)$ to P.
- Upon input $(\texttt{audit}, cm)$ from auditor AU, if $\mathsf{Records}[cm] = \bot$ then return $\bot$. Otherwise, set $(v^{\text{in}}, P, st) \leftarrow \mathsf{Records}[cm]$. If P is not audited by AU, then return $\bot$, else return (v, P).

The protocol is adapted as follows. First, each commitment also contains the identity of the previous owner. This is not technically necessary but allows to prove that the auditable information is correct while keeping the description here compact. The binding between the auditor and the user is achieved through a (structure-preserving) signature from A. The auditing functionality is implemented as follows: A party P that executes a transfer encrypts the following information:

- To its own auditor, for each input the value v^{in} and current owner P. For each output the value v_j^{out}, sender P, and receiver R_j.
- For each output to R_j, to the auditor of R_j the value v_j^{out}, sender P, and receiver R_j.

This is achieved by encrypting the information, including the resulting ciphertexts in the transfer, and proving that the encryption is consistent with the information in the commitment.

For concreteness, consider an input described by commitment $cm = \text{commit}(crs, (v^{\text{in}}, P, P', \rho^{\text{in}}); r_{\text{cm}})$. We encrypt current owner $c_1 = \text{enc}(\mathsf{pk}_{AU}, P; r_1)$ and value $c_2 = \text{enc}(\mathsf{pk}_{AU}, v, r_2)$. Then we generate a NIZK proof:

$$\begin{aligned}\text{PK}\big\{\big(v^{\text{in}}, P, P', \rho^{\text{in}}, \mathsf{Sig}, \mathsf{pk}_{AU}, r_1, r_2\big) :\\ \mathsf{Vf}(\mathsf{pk}_C, (v^{\text{in}}, P, P', \rho^{\text{in}}), \mathsf{Sig}) \wedge \mathsf{Vf}(\mathsf{pk}_A, (P, \mathsf{pk}_{AU}), \mathsf{Sig}_A)\\ \wedge\, c_1 = \text{enc}(\mathsf{pk}_{AU}, P; r_1) \wedge c_2 = \text{enc}(\mathsf{pk}_{AU}, v^{\text{in}}, r_2)\big\}\end{aligned}$$

where pk_C and pk_A are public, and c_1 and c_2 are part of the transaction.

Similarly, for a transfer from P to R and an output commitment $cm = \text{commit}(crs, (v^{\text{out}}, R, P, \rho^{\text{out}}); r_{\text{cm}})$, we encrypt to the auditor (here we use the one of P) the sender $c_1 = \text{enc}(\mathsf{pk}_{AU}, P; r_1)$, the receiver $c_2 = \text{enc}(\mathsf{pk}_{AU}, R; r_2)$, and the value $c_3 = \text{enc}(\mathsf{pk}_{AU}, v^{\text{out}}; r_3)$. We then generate a NIZK proof:

$$\begin{aligned}\text{PK}\big\{\big(v^{\text{out}}, R, P, \rho^{\text{out}}, r_{\text{cm}}, \mathsf{pk}_{AU}, \mathsf{Sig}_A\big) :\\ \text{open}(crs, cm, (v^{\text{out}}, R, P, \rho^{\text{out}}), r_{\text{cm}}) \wedge \mathsf{Vf}(\mathsf{pk}_A, (P, \mathsf{pk}_{AU}), \mathsf{Sig}_A)\\ \wedge\, c_1 = \text{enc}(\mathsf{pk}_{AU}, P; r_1) \wedge c_2 = \text{enc}(\mathsf{pk}_{AU}, R, r_2) \wedge c_3 = \text{enc}(\mathsf{pk}_{AU}, v^{\text{out}}; r_3)\big\}\end{aligned}$$

with public parameters crs and pk_A, as well as cm, c_1, c_2, and c_3 taken from the transaction.

4.6.7 Security Analysis

This section contains the main result of the paper, namely that the protocol in Section 4.6.5 instantiates functionality $\mathcal{F}_{\text{TOKEN}}$.

Theorem 5. *Assume that* $\text{COM} = (\text{ccrsgen}, \text{commit}, \text{open})$ *is a commitment scheme that is perfectly hiding and computationally binding. Assume that* $\text{VRF} = (\text{vkeygen}, \text{eval}, \text{check})$ *is a verifiable random function. Then* π_{TOKEN} *realizes* $\mathcal{F}_{\text{TOKEN}}$ *with static corruption. Corruption is malicious for* I *and users, and honest-but-curious for* C*.* A *is required to be honest, but is inactive during the main protocol phase.*

The restriction that C can only be corrupted in an honest-but-curious model is necessary: Otherwise C can issue signatures on arbitrary commitments, even ones that are not stored in $\mathcal{F}_{\text{LEDGER}}$. We use the composition result of [CDT19] to prove this, since we want to prove correctness of the evaluation of the verification algorithm.

Proof. We use the proof technique of Camenisch et al. [CDT19] in instantiating the functionalities $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{SIG}}$, and $\mathcal{F}_{\text{BLINDSIG}}$ in a way that $\mathcal{F}_{\text{NIZK}}$ can call out to $\mathcal{F}_{\text{SIG}}$ and $\mathcal{F}_{\text{BLINDSIG}}$ for the verification of signatures. This has the advantage that the respective clauses in the statement are ideally verified.

We then need to describe a simulator. Simulator $\mathcal{S}$ emulates functionalities $\mathcal{F}_{\text{LEDGER}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{A-AUTH}}$, $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{SIG}}$, and $\mathcal{F}_{\text{SMT}}$. To emulate $\mathcal{F}_{\text{LEDGER}}$, $\mathcal{S}$ manages an initially empty internal ledger and allows $\mathcal{A}$to read it via `retrieve` or append messages as described below. $\mathcal{S}$ initially sets $\mathit{initialized} \leftarrow \mathsf{false}$. We start by describing the behavior of $\mathcal{S}$ upon outputs provided by $\mathcal{F}_{\text{LEDGER}}$.

- Upon receiving $(\texttt{initialized}, P)$ for $P \in \{A, C\}$, generate a signature key pair for the respective party and simulate the public key of the respective party being registered at $\mathcal{F}_{\text{REG}}$. After receiving this for both A and C, set $\mathit{initialized} \leftarrow \mathsf{true}$.

- Upon receiving $(\texttt{registered}, P)$ from $\mathcal{F}_{\text{TOKEN}}$, mark P as registered and generate output $(\texttt{registered}, P)$ as a message from $\mathcal{F}_{\text{A-AUTH}}$ to $\mathcal{A}$.
- Processing of pending messages (several occasions, see below) for party P: For every record tx marked for delayed processing, proceed as follows.
  - If I is corrupt and $tx = (\texttt{issue}, P', v, cm_0, \psi_0, \psi_2)$, then issue $(\texttt{verify}, y, \psi_0)$ to $\mathcal{A}$as an output of $\mathcal{F}_{\text{NIZK}}$, with $y = (crs, cm_0, v, P')$, and expect as response a witness w. If $w = (r_{\text{cm}}, \rho^{\text{in}})$ is valid for cm_0, and proof ψ_2 is valid according to the simulated instance of $\mathcal{F}_{\text{A-AUTH}}$, then provide the input $(\texttt{issue}, v, cm_0)$ to $\mathcal{F}_{\text{TOKEN}}$.
  - If $tx = (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$, then issue $(\texttt{verify}, y, \psi_1)$ to $\mathcal{A}$as an output of $\mathcal{F}_{\text{NIZK}}$, with $y = (\mathsf{pk}_C, crs, (cm_j)_{j=1}^n, \mathsf{pk}_A, (sn_i)_{i=1}^m)$. Expect as response from adversary $\mathcal{A}$a witness $w = ((\mathsf{Sig}_i, v_i^{\text{in}}, \rho_i^{\text{in}}, \pi_i)_{i=1}^m, P, \mathsf{Sig}_A, (R_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})_{j=1}^n)$. If w is valid, and $\psi_{2,1}, \ldots, \psi_{2,n}$ are valid according to the simulated instance of $\mathcal{F}_{\text{A-AUTH}}$, and corresponding messages have been sent, then mark tx as valid. Provide a request $(\texttt{transfer}, P, (cm_i)_{i=1}^m, (R_j, v_j^{\text{out}}, cm[out]_j)_{j=1}^n)$ to $\mathcal{F}_{\text{TOKEN}}$.
  - For all valid transactions tx where in the meantime the corresponding message $(\texttt{coin}, cm_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ is sent, input $(\texttt{deliver}, cm_j)$ to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{read?}, P)$ from $\mathcal{F}_{\text{TOKEN}}$, process pending messages and return $(\texttt{read!}, P)$ to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{issue}, v)$ from $\mathcal{F}_{\text{TOKEN}}$, first process pending messages. Then, generate a new all-zero commitment $(cm^*, r_{\text{cm}}^*) \leftarrow\!\!\$\ \text{commit}(crs, (0,0,0))$. Next, emulate an output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ for the statement $y = (crs, cm^*, v, P)$ and proceed upon an input $(\texttt{done}, \psi_0^*)$ for the same instance of $\mathcal{F}_{\text{NIZK}}$. Emulate the proof ψ_2^* as in $\mathcal{F}_{\text{A-AUTH}}$, storing the respective instance as a record. Append $(\texttt{issue}, v, cm^*, \psi_0^*, \psi_2^*)$ to the internal ledger. Input $(\texttt{issue}, cm^*)$ to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{transfer}, m, n, L)$ from $\mathcal{F}_{\text{TOKEN}}$, first process pending messages. For each $i = 1, \ldots, m$, generate a random serial number sn_i^*. Then proceed as follows for $j = 1, \ldots, n$. If there is no entry for j in L, then generate a commitment $(cm_j^*, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (0,0,0))$. If there is an entry $(j, P, R_j, v_j^{\text{out}})$, then generate a random value $\rho_j^{\text{out}} \leftarrow\!\!\$\ \{M\}$ and compute commitment $(cm_j^*, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (v_j^{\text{out}}, R_j, \rho_j^{\text{out}}))$. Next, emulate the output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ for instance $y = (\mathsf{pk}_C, crs, (cm_j^*)_{j=1}^n, \mathsf{pk}_A, (sn_i^*)_{i=1}^m)$ and record the proof ψ_1^* returned by $\mathcal{A}$. Emulate the proofs $\psi_{2,1}^*, \ldots, \psi_{2,i}^*$ as in $\mathcal{F}_{\text{A-AUTH}}$. Append transaction $(\texttt{transfer}, (sn_i^*, \psi_{2,i})_{i=1}^m, (cm_j^*)_{j=1}^n, \psi_1^*)$ to the internal ledger and emulate transmission of n messages of the same length as $(\texttt{token}, cm_j^*, r_{\text{cm}}^j, v_j, \rho_j^{\text{out}})$ on $\mathcal{F}_{\text{SMT}}$ (i.e., append the length to the internal queue). Respond with $(\texttt{transfer}, (cm_j^*)_{j=1}^n)$ to $\mathcal{F}_{\text{TOKEN}}$. When $\mathcal{A}$delivers a message on $\mathcal{F}_{\text{SMT}}$, input $(\texttt{deliver}, cm_j^*)$ for the corresponding j to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{transfer}, cm)$, first process pending messages. Record cm in the state of the simulated party C, and proceed as in the above case.
- Upon input $(\texttt{append}, s, P')$ from $\mathcal{A}$for a corrupt P', append s to the ledger and mark for delayed processing. Return to $\mathcal{A}$.

If $\mathcal{S}$ obtains from $\mathcal{A}$a query to $\mathcal{F}_{\text{A-AUTH}}$ in the name of a corrupt party P that is marked as registered, then $\mathcal{S}$ internally handles the inputs `prove` and `verify` just like $\mathcal{F}_{\text{A-AUTH}}$. If $\mathcal{A}$provides an input message x to $\mathcal{F}_{\text{SMT}}$ on behalf of a corrupted party P, then the message is ignored unless it is of the format $x = (\texttt{token}, cm, r_{\text{cm}}, v, \rho)$. If it has the right format, then $\mathcal{S}$ checks whether the corresponding transaction tx exists on $\mathcal{F}_{\text{LEDGER}}$; if it does, then input the respective `deliver` message to $\mathcal{F}_{\text{TOKEN}}$. If such a transaction does not exist, then store the message x for later. Our goal is now to prove that if the commitment and the VRF are secure, then the ideal and real experiments are indistinguishable. We prove this by describing a sequence of experiments, where EXEC_0 is

the real experiments and we transform it step-by-step into the ideal experiment, showing for each adjacent pair of steps that they are indistinguishable. The overall statement then follows via the triangle inequality.

Experiment EXEC_1 is almost the same as EXEC_0 but commitments generated during $(\texttt{issue}, v)$ at an honest party P as well as commitments generated during $(\texttt{transfer}, \dots)$ at an honest party P, where R is also honest, are replaced by commitments generated via $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (0,0,0))$. Functionality $\mathcal{F}_{\text{NIZK}}$ is changed so that it does not actually check the input of honest parties.

Experiments EXEC_0 and EXEC_1 are equivalent since the commitment is perfectly hiding and therefore the distribution of the output to the adversary is unchanged. As all inputs of the honest parties' protocols to $\mathcal{F}_{\text{NIZK}}$ are correct, omitting the checks has no effect.

Experiment EXEC_2 is almost the same as EXEC_1 but serial numbers output by honest parties are replaced by uniformly random values from the same set. Experiments EXEC_2 and EXEC_1 are indistinguishable because of the pseudorandomness of VRF, which is easily proved by reduction. Note that the environment never sees honestly generated proofs.

In the following, we describe the response to environment queries in both EXEC_2 and the ideal experiment and point out the differences. We assume that the state in terms of valid commitments is the same prior to the input, and show when the output to the query is the same and when the state in terms of valid commitments remains consistent. The consistency of the input-output behavior is relatively straightforward to check for most inputs. We focus here on the ones used in `transfer`.

- On input $(\texttt{transfer}, \dots)$ from an honest user P, if not both of P and all $R_1, \dots, R_j$ are registered, then the request is ignored in both cases. Also if not all transferable commitments $cm_1, \dots, cm_i$ exist and are associated to user P, both invocations abort. The protocol π_{TOKEN} then generates new commitments $cm'_1, \dots, cm'_j$ and send them for the proof to $\mathcal{F}_{\text{NIZK}}$, which requests a proof ψ_1 from $\mathcal{A}$. Upon return, π_{TOKEN} generates an additional proof ψ_2 via $\mathcal{F}_{\text{A-AUTH}}$, sends the transaction $(\texttt{transfer}, \dots)$ to $\mathcal{F}_{\text{LEDGER}}$ and the `token` messages to $\mathcal{F}_{\text{SMT}}$. If any R_j is corrupt, this latter invocation means that $\mathcal{A}$learns $(r^j_{\text{cm}}, v^{\text{out}}_j, \rho^{\text{out}}_j)$ as well as the sender P via $\mathcal{F}_{\text{SMT}}$ in addition.

  The functionality $\mathcal{F}_{\text{TOKEN}}$ provides either just m, n—if all $R_1, \dots, R_j$ are honest—or the values $(j, P, R_j, v^{\text{out}}_j)$ for each corrupt R_j. In the first case, $\mathcal{S}$ generates a commitment to all-zero messages and requests the proof ψ_1 from $\mathcal{A}$via $\mathcal{F}_{\text{NIZK}}$-interaction, in the second case $\mathcal{S}$ has all the data available to perform the same computations as the protocol.

  The output distribution is the same since in both cases the commitment is an all-zero commitment and the serial number is uniformly random.

- Processing of pending transactions. For all new (possibly adversarial) transactions on $\mathcal{F}_{\text{LEDGER}}$, the honest parties first attempt to verify the proofs via $\mathcal{F}_{\text{NIZK}}$. For adversarially-generated proofs, the first attempt for each such proof may lead to a message from $\mathcal{F}_{\text{NIZK}}$ requesting the witness from $\mathcal{A}$. The same messages are generated by $\mathcal{S}$, which then records the messages and issues the proper requests to $\mathcal{F}_{\text{TOKEN}}$. (Note that this processing in $\mathcal{F}_{\text{TOKEN}}$ takes place at this point in time, but the timing is indistinguishable from that in the protocol as each honest user input leads to that user processing the pending transactions in the protocol.)

  For (passively) corrupt C, if a commitment cm is delivered to the receiver, the simulator learns the receiver's identity and emulates the behavior in the real protocol where C also learns both the commitment and the identity of the receiver.

  For adversarial transfers sent to the party via $\mathcal{F}_{\text{SMT}}$, it may mean that the message sent on $\mathcal{F}_{\text{SMT}}$ is not proper (so it is ignored by both π_{TOKEN} and $\mathcal{S}$), or that it parses correctly does not have a corresponding transaction in $\mathcal{F}_{\text{LEDGER}}$ (in the sense that the commitment cm^* in the message does not exist there—then it is also ignored), or that both message and transaction can be found, in which case the view of the party changes when the tokens are found.

The only difference between the two above executions is when the adversary fabricates a transaction in the name of a corrupt party that makes a state transition that is different from the one that is done in $\mathcal{F}_{\text{TOKEN}}$. Let

us first consider `issue` transactions, where the statement is $y = (crs, cm^*, v, P')$. When an honest party verifies the proof with $\mathcal{F}_{\text{NIZK}}$, then $\mathcal{A}$has to provide a proper witness (r^*_{cm}, ρ) such that the commitment opens to $\text{open}(crs, cm^*, (v, P, \rho^*), r^*_{\text{cm}}) = \mathsf{true}$.

Consider a transaction $tx = (\texttt{transfer}, (sn^*_i, \psi_{2,i})^m_{i=1}, (cm^*_j)^m_{j=1}, \psi^*_1)$ input by the adversary. When ψ^*_1 is verified by the honest party, then $\mathcal{A}$is given the statement $y = (\mathsf{pk}_C, crs, (cm^*_j)^n_{j=1}, \mathsf{pk}_A, (sn^*_i)^m_{i=1})$ and provides a witness $w = ((\mathsf{Sig}_i, v^{\text{in}}_i, \rho^{\text{in}}_i, \pi_i)^m_{i=1}, P, \mathsf{Sig}_A, (R_j, r^j_{\text{cm}}, v^{\text{out}}_j, \rho^{\text{out}}_j)^n_{j=1})$, which satisfies the PK-statement

$$
\begin{aligned}
\text{PK}\big\{ \big((\mathsf{Sig}_i, v^{\text{in}}_i, \rho^{\text{in}}_i, \pi_i)^m_{i=1}, P, \mathsf{Sig}_A, (R_j, r^j_{\text{cm}}, v^{\text{out}}_j, \rho^{\text{out}}_j)^n_{j=1}\big) : \\
\forall i \in \{1, \ldots, m\} : \mathsf{Vf}(\mathsf{pk}_C, (v^{\text{in}}_i, P, \rho^{\text{in}}_i), \mathsf{Sig}_i) \\
\wedge \forall j \in \{1, \ldots, n\} : \text{open}(crs, cm_j, (v^{\text{out}}_j, R_j, \rho^{\text{out}}_j), r^j_{\text{cm}}) \\
\wedge \mathsf{Vf}(\mathsf{pk}_A, (P, vpk), \mathsf{Sig}_A) \\
\wedge \forall i \in \{1, \ldots, m\} : \text{check}(vpk, \rho^{\text{in}}_i, sn_i, \pi_i) \\
\wedge \sum_{i=1}^{m} v^{\text{in}}_i = \sum_{j=1}^{m} v^{\text{out}}_j \wedge \forall j \in \{1, \ldots, n\} : 0 \le v^{\text{out}}_j \le \mathsf{max}\big\}.
\end{aligned}
$$

As $\mathsf{Vf}(\mathsf{pk}_C, (v^{\text{in}}_j, P, \rho^{\text{in}}_i), \mathsf{Sig}_i)$ are evaluated via $\mathcal{F}_{\text{BLINDSIG}}$, and C checks the correctness of crs, we also know that $\mathsf{Sig}_1, \ldots, \mathsf{Sig}_m$ were generated for inputs $(\texttt{request}, crs, r^i_{\text{cm}}, (v^{\text{in}}_i, P, \rho^{\text{in}}_i))$, and the commitment $cm^*_i = \text{commit}(crs,$
$(v^{\text{in}}_i, P, \rho^{\text{in}}_i); r^i_{\text{cm}})$ indeed exists on the ledger. Then either cm^i was created during a previous transaction with the same input $(v^{\text{in}}_i, P, \rho^{\text{in}}_i)$ or we can turn $\mathcal{Z}$ into an adversary that breaks the binding property of COM.

As $\mathsf{Vf}(\mathsf{pk}_A, (P, vpk), \mathsf{Sig}_A)$ is evaluated via a call to $\mathcal{F}_{\text{SIG}}$, and the correctness of both $\mathcal{F}_{\text{SIG}}$ and the honesty of A implies that vpk is the *unique* VRF public key associated to P. So at this point we know that vpk and ρ^{in}_i are correct. As $\text{check}(vpk, \rho^{\text{in}}_i, sn^*_i, \pi_i) = \mathsf{true}$, either $(sn^*_i, \_) \leftarrow \text{eval}(vsk, \rho^{\text{in}}_i)$ or we can turn $\mathcal{Z}$ into an adversary against the soundness of VRF. This means that sn^*_i is also correct, no double-spending occurred. The last two lines mean that the sum of all output values and the sum of all input values are the same, so the overall value is preserved (and the input provided by the simulator is accepted by $\mathcal{F}_{\text{TOKEN}}$).

The construction has negligible correctness error due to collision of sequence numbers. □

4.7 Instantiation

In this section, we provide details on how to instantiate the protocol using well-established primitives that do not require any complex setup assumptions.

4.7.1 Pedersen Commitments

The commitment scheme is instantiated with Pedersen commitments [Ped91b] on multiple values. Consider a group $\mathcal{G}$ and generators $g_0, g_1, \ldots, g_\ell \in \mathcal{G}$ such that the relative discrete logarithms between the g_i are not known. A commitment to a vector $(x_1, \ldots, x_\ell) \in \{1, \ldots, |\mathcal{G}|\}^\ell$ of inputs is computed by choosing a uniformly random $r \in \{1, \ldots, |\mathcal{G}|\}$ and computing $(cm, r_{\text{cm}}) \leftarrow (g^r_0 g^{x_1}_1 \cdots g^{x_\ell}_\ell, r)$. Pedersen commitments are perfectly hiding and computationally binding under the discrete-logarithm assumption in group $\mathcal{G}$.

4.7.2 Pointcheval-Sanders (PS) Signatures

We use the signature scheme of Pointcheval and Sanders [PS16] to implement the blind signature used for token certification. The scheme operates in an asymmetric pairing setting with groups $\mathcal{G}_1$ and $\mathcal{G}_2$ of size p, with target group $\mathcal{G}_{\text{T}}$ and bilinear map $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_{\text{T}}$. Key generation skeygen chooses $\tilde{g} \in \mathcal{G}_2$ and $(x, y_1, \ldots, y_\ell) \in \mathbb{Z}^{\ell+1}_p$ and sets $\mathsf{sk} \leftarrow (x, y_1, \ldots, y_\ell)$ and $\mathsf{pk} \leftarrow (\tilde{g}, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_\ell}) = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$.

A signature $\mathsf{Sig} = \mathrm{sign}(\mathsf{sk}, (m_1, \ldots, m_\ell))$ on message vector $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$ is computed as $\mathrm{sign} \leftarrow (h, h^{x+\sum_j y_j m_j})$ with h is randomly-chosen in $\mathcal{G}_1$. Verification of signature $\mathsf{Sig} = (\mathsf{Sig}_1, \mathsf{Sig}_2)$ is performed by checking $\mathsf{Sig}_1 \neq 1_{\mathcal{G}_1}$ and $e(\mathsf{Sig}_1, \tilde{X} \prod_j \tilde{Y}_j^{m_j}) = e(\mathsf{Sig}_2, \tilde{g})$.

PS signatures are CMA under an interactive computational assumption. In follow-up work, Pointcheval and Sanders [PS18] showed that the scheme can be modified to be secure under a non-interactive assumption, by adding and signing another random element m_0. For simplicity, we use the original version in this instantiation.

4.7.3 Certification through Blind Signatures

The functionality $\mathcal{F}_{\text{BLINDSIG}}$ is instantiated by the following protocol π_{BLINDSIG}, which operates in the $\{\mathcal{F}_{\text{NIZK}}, \mathcal{F}_{\text{REG}}, \mathcal{F}_{\text{SMT}}\}$-hybrid model. Let $H_{\mathcal{G}} : \mathcal{G}_1 \to \mathcal{G}_1$ denote a cryptographic hash function modeled as a random oracle.

- Upon input `init`, certifier C generates a new key pair $(\mathsf{sk}, \mathsf{pk})$ with $\mathsf{sk} = (x, y_1, \ldots, y_\ell)$ and $\mathsf{pk} = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$, and sends $(\texttt{register}, \mathsf{pk})$ to $\mathcal{F}_{\text{REG}}$.
- Upon input `pubkey`, P sends $(\texttt{query}, C)$ to $\mathcal{F}_{\text{REG}}$ and outputs the result.
- Upon input $(\texttt{request}, crs, r_{\text{cm}}, (m_1, \ldots, m_\ell))$, proceed as follows.
  1. Pick $z \leftarrow\!\!\$\ \mathbb{Z}_p$ and compute $u \leftarrow g^z$. Compute $cm \leftarrow g_0^{r_{\text{cm}}} \prod_{i=1}^{\ell} g_i^{m_i}$ and $h \leftarrow H_{\mathcal{G}}(cm)$.
  2. For each $i = 1, \ldots, \ell$, choose $r_i \leftarrow\!\!\$\ \mathbb{Z}_p$, $a_i \leftarrow u^{r_i}$, and $b_i \leftarrow h^{m_i} g^{r_i}$.
  3. Obtain the proof
  $$\zeta \leftarrow \mathrm{PK}\{((m_i, r_i)_{i=1}^{\ell}, r_{\text{cm}}) : \bigwedge_{i=1}^{\ell}(a_i = u^{r_i} \wedge b_i = h^{m_i} g^{r_i}) \wedge cm = g_0^{r_{\text{cm}}} \prod_{i=1}^{\ell} g_i^{m_i}\}$$
  on input $(\texttt{prove}, y, w)$ at $\mathcal{F}_{\text{NIZK}}$ with $y = (cm, h, u, a_1, \ldots, a_\ell, b_1, \ldots, b_\ell)$ and $w = ((m_i, r_i)_{i=1}^{\ell}, r_{\text{cm}})$.
  4. Call $\mathcal{F}_{\text{SMT}}$ with $(\texttt{send}, C, (\zeta, crs, cm, u, (a_i, b_i)_{i=1}^{\ell}))$.
- Upon receiving $(\texttt{sent}, P, (\zeta, crs, cm, u, (a_i, b_i)_{i=1}^{\ell}))$ from $\mathcal{F}_{\text{SMT}}$, certifier C proceeds as follows:
  1. Verify ζ via $\mathcal{F}_{\text{NIZK}}$ and compute $h \leftarrow H_{\mathcal{G}}(cm)$. If verification fails, input $(\texttt{send}, P, \perp)$ to $\mathcal{F}_{\text{SMT}}$ and stop.
  2. Store $(\zeta, crs, cm, u, (a_i, b_i)_{i=1}^{\ell}, h)$ internally and output to signer C message $(\texttt{request}, P, crs, cm)$.
- Upon input $(\texttt{sign}, cm)$, signer C proceeds as follows:
  1. If no record for commitment cm is stored, stop.
  2. Compute $\bar{b} \leftarrow h^x \prod_{i=1}^{\ell} b_i^{y_i} = g^{\bar{r}} h^x \prod_{i=1}^{\ell} h^{m_i y_i}$ and $\bar{a} \leftarrow \prod_{i=1}^{\ell} a_i^{y_i} = u^{\bar{r}}$.
  3. Call $\mathcal{F}_{\text{SMT}}$ with $(\texttt{send}, P, (\bar{a}, \bar{b}))$.
- Upon receiving $(\texttt{sent}, C, \bar{m})$ from $\mathcal{F}_{\text{SMT}}$, receiver P proceeds as follows:
  1. If $\bar{m}$ cannot be parsed as $(\bar{a}, \bar{b}) \in \mathcal{G}_1^2$ output $(\texttt{result}, \perp)$ and stop.
  2. Compute $h' \leftarrow \bar{b}\bar{a}^{-1/z}$ and check $e(h, \tilde{X} \prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}) \stackrel{?}{=} e(h', \tilde{g})$. If the check fails, output $(\texttt{result}, \perp)$ and stop. Else output $(\texttt{result}, (h, h'))$.

Note that in the above description we use a deterministic variant of Pointcheval-Sanders signatures. Namely, generator h is not selected randomly in $\mathcal{G}_1$, rather it is computed as the hash of commitment cm. The reason behind this slight modification is to enable a non-interactive distributed signature (i.e. signers do not need to interact), see Section 4.7.7 for further details. It is easy to show that the security of this variant holds in the random-oracle model.

Lemma 4. *Protocol π_{BLINDSIG} realizes $\mathcal{F}_{\text{BLINDSIG}}$ under Assumption 2 of Pointcheval and Sanders [PS16], given that C is honest-but-curious and $\mathcal{A}$does not have access to the secret key of C.*

A similar protocol has been provided as part of the Coconut systems by Sonnino, Al-Bassam, Bano, Meiklejohn, and Danezis [SABB+18], but the protocol there is slightly less efficient. Furthermore, Section 4.7.7 shows how the above token certification can be distributed.

4.7.4 Groth Signatures

We use Groth's structure preserving signatures [Gro15] to bind a user public key to an auditor public key. The signature scheme operates in a pairing setting with groups $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_{\text{T}}$, and on messages in $\mathcal{G}_1$. Let g and $\tilde{g}$ be random generators of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. Key generation $\text{skeygen}(\lambda, \ell)$ selects a vector $\mathsf{sk} = (x, y_1, \ldots, y_{\ell-1}) \leftarrow\!\!\$\ \mathbb{Z}_p^\ell$ and a random generator $h \leftarrow\!\!\$\ \mathcal{G}_1$, and computes $\mathsf{pk} \leftarrow (h, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_{\ell-1}) = (h, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_{\ell-1}})$. Signature $\text{sign}(\mathsf{sk}, (m_1, \ldots, m_\ell))$ selects uniformly at random $r \leftarrow\!\!\$\ \mathbb{Z}_p$, computes $\tilde{a} \leftarrow \tilde{g}^{1/r}$, $b \leftarrow (hg^x)^r$, and $c \leftarrow (h^x m_\ell \prod_{i=1}^{\ell-1} m_i^{y_i})^r$, and sets $\mathsf{Sig} \leftarrow (\tilde{a}, b, c)$. Verification of signature $\mathsf{Sig} = (\tilde{a}, b, c)$ for messages $(m_1, \ldots, m_\ell)$ proceeds by verifying two pairing equations $e(b, \tilde{a}) = e(h, \tilde{g})e(g, \tilde{X})$ as well as

$$e(c, \tilde{a}) = e(h, \tilde{X})e(m_\ell, \tilde{g}) \prod_{i=1}^{\ell-1} e(m_i, \tilde{Y}_i).$$

Dodis-Yampolskiy VRF

We use the VRF of Dodis and Yampolskiy [DY05] that operates in the pairing setting. Key generation $\text{vkeygen}(\lambda)$ chooses a random $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p$ and sets $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$. Evaluation $\text{eval}(\mathsf{sk}, x)$ aborts if $\mathsf{sk} + x \notin \mathbb{Z}_p^\times$. It computes output $y \leftarrow e(g, \tilde{g})^{1/(\mathsf{sk}+x)} \in \mathcal{G}_{\text{T}}$ and proof $\pi \leftarrow \tilde{g}^{1/(\mathsf{sk}+x)} \in \mathcal{G}_2$. Verification $\text{check}(\mathsf{pk}, x, y, \pi)$ checks whether $e(g, \pi) = y$ and $e(\mathsf{pk} \cdot g^x, \pi) = e(g, \tilde{g})$; if so it outputs $b = 1$.

4.7.5 Groth-Sahai NIZK

Since all equations we have to verify — for the Pointcheval-Sanders signatures, the Pedersen commitments, the Groth signatures, and the Dodis Yampolskiy VRF — are defined in terms of bilinear groups, we propose to use Groth-Sahai proofs [GS08] to instantiate $\mathcal{F}_{\text{NIZK}}$ in our solution.

4.7.6 ElGamal Public-Key Encryption

We use ElGamal encryption [ElG85a]. Key generation $\text{ekeygen}(\lambda)$ chooses a uniformly random exponent $\mathsf{sk} \leftarrow\!\!\$\ \{1, \ldots, |\mathcal{G}|\}$ and computes $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$. Encryption $\text{enc}(\mathsf{pk}, m)$ chooses a uniformly random $r \leftarrow\!\!\$\ \{1, \ldots, |\mathcal{G}|\}$ and computes $c \leftarrow (g^r, \mathsf{pk}^r m)$. Decryption $\text{dec}(\mathsf{sk}, c)$ with $c = (c_1, c_2)$ computes $m \leftarrow c_2 c_1^{-\mathsf{sk}}$. The encryption scheme is semantically secure and key private under the Decisional Diffie-Hellman assumption.

4.7.7 Range Proofs

Our protocol requires range proofs to ensure that no field wrap-arounds are exploited to increase the quantity of tokens in a transfer. The range proof we use is based on the work of Camenisch, Chaabouni, and shelat [CCas08], instantiated with Pointcheval-Sanders signatures.

Distributing certification

The Pointcheval-Sanders signature scheme can be extended into a non-interactive t-out-of-n threshold signature scheme. Consider n signers $C_1, \ldots, C_n$ from which a recipient P collects at least t signature shares that can be combined into a complete signature. We describe the process with a trusted key generation, however, notices

that it is straightforward to convert the key generation mechanism into a multiparty computation between the signers (see e.g. [GJKR07]). We describe the key generation algorithm $\mathrm{tkeygen}$ and the reconstruction algorithm $\mathrm{combine}$. The algorithm to produce a signature share is identical to original signing algorithm (taking secret key share as input instead of the overall secret key). That is, to sign a message $(m_1, \ldots, m_\ell)$, signer C_i calls algorithm $(h, h') \leftarrow\!\!\$\ \mathrm{sign}(\mathsf{sk}_i, (m_1, \ldots, m_\ell)))$ with $\mathsf{sk}_j = (x_j, y_{1j}, \ldots, y_{\ell j})$. The resulting signature share is a valid Pointcheval-Sanders signature for public key $\mathsf{pk}_j = (\tilde{X}_j, \tilde{Y}_{1j}, \ldots, \tilde{Y}_{\ell j})$.

Algorithm $\mathrm{tkeygen}(\lambda, n, t, \ell)$ computes $(\mathsf{sk}_j, \mathsf{pk}_j)_{j=1}^n, \mathsf{pk}$ as follows:

- Pick $\ell + 1$ random polynomials $p_x, p_{y_1}, \ldots p_{y_\ell}$ of degree $t - 1$ with coefficients from $\mathbb{Z}_p$.
- Compute $\tilde{X} \leftarrow g^{p_x(0)}, \tilde{Y}_1 \leftarrow g^{p_{y_1}(0)}, \ldots, \tilde{Y}_\ell \leftarrow g^{p_{y_\ell}(0)}$.
- Compute all $\tilde{X}_j = g^{x_j}$ and $\tilde{Y}_{ij} \leftarrow g^{y_{ji}}$.
- Set $\mathsf{pk} = (\tilde{X}, \tilde{Y}_1 = g^{p_{y_1}(0)}, \ldots, \tilde{Y}_\ell = g^{p_{y_\ell}(0)})$, and $\mathsf{pk}_j = (\tilde{X}_1, \tilde{Y}_{01}, \ldots, \tilde{Y}_{\ell 1})$. Set $\mathsf{sk}_j = (p_x(j), p_{y_0}(j), \ldots, p_{y_\ell}(j))$ and output $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n, \mathsf{pk}_1, \ldots, \mathsf{pk}_n, \mathsf{pk})$.

Algorithm $\mathrm{combine}$, on input $\{(\mathsf{Sig}_i, \mathsf{pk}_i)\}_{i \in S}, (m_1, \ldots, m_\ell)$, for a set $S \subseteq \{1, \ldots, n\}$ with $|S| = t$, proceeds as follows.

- Output $\perp$ if not all $\{(\mathsf{Sig}_i, \mathsf{pk}_i)\}_{i \in S}$ with $\mathsf{Sig}_i = (h_i, h_i')$ have the same h and if $\mathsf{Vf}((\tilde{X}_i, \tilde{Y}_{1i}, \ldots, \tilde{Y}_{\ell i}), (m_1, \ldots, m_\ell), (h,$ does not hold for all $i \in S$.
- Compute Lagrange coefficients $\lambda_j = \prod_{i \in S \setminus j} \frac{i}{i-j}$ for all $j \in S$.
- Compute and output $(h, h' = \prod_{j \in S} {h_j'}^{\lambda_j})$.

Protocol π_{BLINDSIG} from Section 4.7.3 has to be modified as follows:

- Instead of generating a key locally at C, all signers $C_1, \ldots, C_n$ together use $\mathcal{F}_{\text{DKG}}$ to generate the set of keys. Signer C_1 registers the public key pkat $\mathcal{F}_{\text{REG}}$.
- Requestor P sends the request message to parties $C_1, \ldots, C_n$ until it has collected t signatures that verify. It then uses $\mathrm{combine}$ to combine that into a single signature that verifies relatively to pk.

Theorem 6. *Let $n \in \mathbb{N}$ and $t < n$. The above-described variant of the protocol realizes the threshold variant of $\mathcal{F}_{\text{BLINDSIG}}$.*

No further adaptations to the users' protocol beyond the use of the threshold functionality are necessary, as the verification equation for the signatures remains the same.

4.8 Implementation and Performance

To evaluate the feasibility of our protocol, we implemented a prototype using the primitives described in Appendix 4.7 and measured its performance. By design, our prototype is compatible with Hyperledger Fabric and requires minimal changes to be integrated. This section elaborates on the integration effort and measures the overhead incurred by our scheme.

4.8.1 Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain system. Hyperledger Fabric entities exchange messages, called *transactions*, over the Hyperledger Fabric network. A transaction is used to introduce either a new smart contract (*chaincode* in Hyperledger Fabric terms) into the system or changes to the state of an already existing chaincode (i.e. *execute*). The first process is called *chaincode instantiation*, whereas the latter process is referred to as

chaincode invocation. A special type of transactions, *reconfiguration transactions*, is used to introduce changes to the system configuration.

In a Hyperledger Fabric network, we identify three types of participants: (i) *clients* who submit transactions to the network in order to instantiate or invoke chaincodes, or to reconfigure the system; (ii) *peers* which execute chaincodes, validate transactions and maintain a (consistent) copy of the ledger; and (iii) *orderers* which jointly decide the order in which transactions would appear in the ledger.

For the proper operation of the system, each instance of Hyperledger Fabric considers one or more membership service providers (in short, MSPs) that issue long-term identities to parties falling under their authority. These identities allow system entities to securely interact with each other; essentially, MSPs provide the required abstractions to validate identities; namely, to compute and verify signatures. The configuration of valid MSPs is included in the genesis block of each Hyperledger Fabric instance and can be updated via reconfiguration transactions.

Hyperledger Fabric follows an *execute-order-validate* model. Here, chaincodes are speculatively *executed* on one or more peers upon a client request—called *chaincode proposal*—prior to submitting the resulting transaction for ordering. Execution results are signed by the peers that generated them in *chaincode endorsements* and are returned to the client who requested them. Endorsements (i.e. peer signatures) are included in the transaction that the client constructs and sends to the ordering service. The latter *orders* the transactions it receives and outputs a first version of the ledger called *raw ledger*. Raw ledger is provided to the peers of the network upon demand. Upon receiving the raw ledger, peers *validate* the ordered transactions against the *endorsement policy* of their origin chaincodes. An endorsement policy specifies the endorsements that a transaction should carry to be deemed valid. If validation completes successfully, then the transaction is committed to the ledger.

Notice that although there is a separation in Hyperledger Fabric between clients and peers, there still is a communication channel between the two, leveraged by clients to acquire endorsements on the chaincodes they wish to invoke and perform queries on the ledger state. In the following section, we show how to make use of this channel to extend Hyperledger Fabric with our protocol.

4.8.2 Integration Architecture

We first require that each issuer, user and auditor operates a Hyperledger Fabric client. These clients are used to generate an `issue` or `transfer` transactions, submit token certification requests and read from the ledger. Along these lines, we outsource the cryptographic operations required to generate token transactions to a *prover chaincode* in the aim of alleviating the load at the client. This setting assumes that each client possesses a peer that she trusts with the computation of the zero-knowledge proofs and serial numbers. We contend that this is a reasonable assumption especially for Hyperledger Fabric that focuses on enterprise applications.

We also make use of the already-existing communication protocol between the clients and the peers to implement what we call, for convenience, *certifier chaincode*. This is a chaincode that runs only on a selective set of peers chosen at setup time and trusted to jointly certify valid tokens, following the protocol in Section 4.7.7. Each such a peer is endowed with a share of the certification signing key, and whenever invoked, provides its share to the certifier chaincode.

Finally, we leverage the membership service infrastructure of Hyperledger Fabric to grant long-term identities to issuers and users. In particular, we integrate the identity mixer MSP of Hyperledger Fabric with our solution to allow privacy preserving user authentication. When it comes to assigning auditors to users, we use an off-band channel to bind identity mixer user identities with auditor encryption public keys. In a real implementation, this could be accommodated by an external identity management service, preferably distributed[1].

Notice that our protocol uses the ledger only as a time-stamping service, without any validation functionalities; those are offloaded indirectly to certifiers and auditors. This could be supported in Hyperledger Fabric directly by setting the endorsement policy of the prover chaincode to `any`. We note that we plan to extend our prototype to allow the ledger to also validate token transactions. More concretely, we intend to exploit the fact

[1]The auditor assignment requires structure preserving signatures, which as of now lack single-round distributed instantiations.

	token certification	
user	$198,90$	
certifier	$123,36$	
	proof generation	**proof validation**
overall computation	$1992,844$	$2885,134$
in-out consistency	$157,43$	$287,21$
token validity	$263,02$	$322,34$
serial number	$208,24$	$452,55$
auditability	$1361,99$	$1823,01$

Table 4.1: This table shows the performance numbers of token certification and transfer in milliseconds (ms). We note that the transaction size is a little over 63KB; this figure however can be further optimized.

that Hyperledger Fabric supports pluggable transaction validation [Hyp] that allows chaincodes to specify their own custom validation rules, in our case, the custom validation would consist of executing the verification of the ZK proofs.

4.8.3 Performance Numbers

We installed Hyperledger Fabric client and peer infrastructure with our custom validation process on a MacBook Pro (15-inch, 2016), with 2.7 GHz Intel Core i7, and 16 GB of RAM. We implemented our prototype in golang, as this is the core language of Hyperledger Fabric, and used EC groups in BN256 curves. We instantiated both prover and certifier chaincodes on all the peers in the network, while disseminating the secret shares needed for token certification only to the peers reserved for that purpose. For efficiency reasons, we used Schnorr [Sch91] proofs to implement some of the zero-knowledge proofs; this however comes at the cost of formally losing composable security.

We measured the time required to produce and validate a `transfer` transaction as these operations are the most computationally-heavy. We produced our results using the measurements of 100 runs of each operation.

Our results are shown in Table 4.1 for transfers with two inputs and two outputs. Although our scheme supports an arbitrary number of inputs and outputs, we opt for this combination as it is the common configuration in existing schemes. We assume that there is one certifier and that the maximum liquidity that can be issued or transferred at anytime is capped at 2^{16}.

In the performance evaluation of transaction generation and validation, we present separately the overhead resulting from (i) checking that the input and outputs preserve value and type; (ii) hiding the transaction graph, cf. entries token validity and serial numbers; (iii) and auditability. Our measurements show that the overall transaction construction time is little less than 2s, whereas transaction validation takes a little less than 3s. Auditability is the most expensive operation as it requires the generation and the verification of multiple proofs of correct encryption under obfuscated public keys; more than 2/3 of the overall computation time. Second comes the operations that hide the transaction graph with proof generation time of almost 0.5s and verification time of roughly 0.7s. This shows that in applications where auditability and full privacy are not a priority, our solution performs relatively-well, less than 158ms for transaction generation and 287ms for its verification. Our performance figures exclude proofs of ownership as the performance of those is outweighed by the Identity Mixer overhead.

While these numbers are not yet favorable to a wide adoption, we would like to stress that the AMCL library underlying our implementation is not optimized. An optimization in the crypto libraries is expected to bring in a speedup of at least one order of magnitude [Kra15]. We also note that the current implementation did not investigate possibilities of parallelization.

We also measured the time it takes to get a token certificate. Table 4.1 shows that the computation at the user takes around 199ms, whereas the overhead at the certifier is 123ms.

Chapter 5

Timed Cryptographic Primitives

Timestamping is an important cryptographic primitive with numerous applications. The availability of a decentralized blockchain such as that offered by the Bitcoin protocol offers new possibilities to realise timestamping services. Nevertheless, to our knowledge, neither the classical timestamping results, nor recent blockchain-based proposal are formally defined and proved in a composable setting. In this chapter, we continue the work stated in the deliverable D2.2, and show how to UC-realize the functionalities proposed in Chapter 5 of D2.2. In the first part of the chapter we provide a high level overview of the functionalities and their UC-realization. In the remaining part of the chapter we formally show how to realize these functionalities. For a formal description on the UC-functionalities considered in this chapter we refer the reader to Chapter 5 of the Deliverable D2.2.

5.1 Technical Overview

In this section we provide a technical summary of our contributions. In the later sections, we provide elaborated versions of them.

5.1.1 Beacon functionality and enhanced ledger

In order to construct a source of sufficiently unpredictable and publicly verifiable randomness, we design and use a blockchain-based beacon. We investigate how an ideal beacon can be weakened so that it is implementable by a protocol which uses the ledger functionality and a random oracle. In particular, we specify a weak beacon functionality which is sufficiently strong to be used for timestamping cryptographic primitives. Our beacon, similar to [BMTZ17, BGK+18, GKL15, PSS17], relies on the assumption that the blocks generated by the honest parties include at least $\hat{\lambda}$ bits entropy. However, this does not mean that it is possible to extract at least $\hat{\lambda}$-bit randomness from a sequence of blocks that contains an honestly generated entry. Generally speaking, the reason is that parties work in parallel to extend the chain, and there is a possibility that they collide which gives the adversary the choice between the colliding blocks. This gives the adversary a bit more power in guessing the output of the beacon. Informally, the entropy of the honest block can be reduced by a factor that depends on the number of honest blocks proposed within a small window from the round in which the beacon emits its value. Nevertheless, as we will argue later, this issue can at most eliminate a few bits of entropy from the beacon. Attempting to capture the above, we hit a shortcoming of the ledger from [BMTZ17]. The reason is that the current definition of the ledger does not account for the entropy of the honest blocks. A way to rectify that would be to change the ledger functionality in a *non-black-box manner* and reprove the its security. To resolve the aforementioned issue, we introduce an explicit wrapper called WBU-wrapper. In a nutshell, the WBU-wrapper wraps the ledger functionality, i.e., takes control of all its interfaces, and acts as an upper relayer. Together with the formal definition of the WBU-wrapper we also show that the (UC-abstraction of the) Bitcoin backbone protocol in [BMTZ17] emulates the wrapped ledger. As a next step we define a *weak* beacon functionality $\mathcal{B}^{\mathsf{w}}$ and provide an instantiation of it using the wrapped ledger functionality. This functionality can be queried with a

round number ρ, and return the couple $(\eta, \mathtt{tsl})$, where η denotes the random value, and $\mathtt{tsl}$ represents the output number (i.e., there is not a one-to-one correspondence between the rounds of $\mathcal{G}_{\mathtt{clock}}$ and the output number of $\mathcal{B}^{\mathsf{w}}$). Note that any implementation of an ideal randomness beacon is expected to meet (at least) the following requirements:

Agreement on the Output: The output of the beacon can be verified by any party who has access to the beacon.

Liveness: The beacon generates new values as time advances. The output of the beacon can be verified (albeit at some point in the future) by any party who has access to the beacon.

Perfect Unpredictability: No one should be able to bias or even predict (any better than guessing) the outcome of the beacon before it has been generated.

Nevertheless, due to the adversarial influence on the contents of the ledger, we cannot obtain a perfect beacon from the ledgers that are implemented by common cryptocurrencies (cf. also [BGZ16b] for an impossibility). Indeed, we will allow the adversary to predict the next Δ outputs of $\mathcal{B}^{\mathsf{w}}$, and to generate a new output after at most R rounds. At a high level, our beacon protocol works as follows. A party that wants to compute the latest beacon's output simply needs to compute the hash of the latest $\ell - \mu + 1$ blocks of the ledger. This ensures that at least one hashed block is honest, and therefore that the adversary cannot predict more than the next Δ outputs, where $\Delta = \ell - \mu$. Moreover, $R = \mathtt{MaxRound}$, where $\mathtt{MaxRound}$ denotes the maximum number of rounds after that the state of the ledger has to be extended. We note passing that one might be tempted to implement the $\mathcal{B}^{\mathsf{w}}$ by hashing only the last (stable) block of the hash chain, which would yield to a more efficient construction. However, as we will argue later, this approach is not generic and is suitable only for a certain type of blockchains. For more details on the wrapper and on the weak beacon we refer the reader to Section 5.2 and 5.3 respectively.

5.1.2 Timed digital signature

In this section, we provide a technical overview of our timed signatures. Here, we extend the standard notion of the digital signature by different levels of timing guarantees. In our model, a timestamped signature σ for a message m is equipped with a time mark τ that contains information about when σ was computed by the signer (σ corresponds to an output of $\mathcal{G}_{\mathtt{clock}}$). We refer to this special notion of signature as *Timed Signature (TSign)*. We define three categories of security for TSign: *backdate* security, *postdate* security, and their combination which we refer to just as *timed security*. Intuitively, backdate security guarantees that the signature σ time-marked with τ has been computed some time *before* τ; postdate security guarantees that the signature σ was computed some time *after* τ; and timed security provides to the party that verifies the signature σ a time interval around τ in which σ was computed. We will formally define these three new security notions with a single notion $\mathcal{F}_{\sigma}^{\mathsf{w},\mathtt{t}}$ parameterized by a flag $\mathtt{t} \in \{+, -, \pm\}$ where $\mathtt{t} = \text{“}-\text{”}$ indicates that the functionality guarantees backdate security, $\mathtt{t} = \text{“}+\text{”}$ indicates postdate security, and $\mathtt{t} = \text{“}\pm\text{”}$ indicates timed security. Analogously to the weak beacon, $\mathcal{F}_{\sigma}^{\mathsf{w},\mathtt{t}}$ and all parties that have access to this functionality, are registered to $\mathcal{G}_{\mathtt{clock}}$ which provides the notion of time inherently required by our model.

In a nutshell, and more formally, the functionality $\mathcal{F}_{\sigma}^{\mathsf{w},\mathtt{t}}$ provides to its registered parties a new time-slot $\mathtt{tsl} \in \mathbb{N}$ every R rounds (in the worst case). Once a time slot $\mathtt{tsl}$ is issued, it can be used to time(stamp) a signature σ. The meaning of $\mathtt{tsl}$ depends on the notion of security that we are considering. For backdate security (i.e., $\mathtt{t} = \text{“}-\text{”}$), a signature σ marked with $\mathtt{tsl}$ denotes that σ was computed during a time slot $\mathtt{tsl}' \leq \mathtt{tsl}$. For postdate security ($\mathtt{t} = \text{“}+\text{”}$) $\mathtt{tsl}$ denotes that σ was computed during a time slot $\mathtt{tsl}' \geq \mathtt{tsl}$. For timed security, the signature σ is equipped with two time-marks $\mathtt{tsl}_{\mathtt{back}}$ and $\mathtt{tsl}_{\mathtt{post}}$ that denote that σ was computed in a time-slot $\mathtt{tsl}'$ such that $\mathtt{tsl}_{\mathtt{post}} \leq \mathtt{tsl}' \leq \mathtt{tsl}_{\mathtt{back}}$. A new time-slot issued by $\mathcal{F}_{\sigma}^{\mathsf{w},\mathtt{t}}$ can be immediately seen and used by $\mathcal{A}$. However, the adversary can delay honest parties from seeing new time-slots—i.e., truncate the view that each honest party has of the available time-slots. That is, for each party p_i, $\mathcal{A}$ can decide to *hide* the most recent W-many available time-slots. There are also other subtleties that the ideal functionality $\mathcal{F}_{\sigma}^{\mathsf{w},\mathtt{t}}$ needs to capture and this makes the functionality more complicated that one might expect. We refer the reader to the formal definition of the functionality in Chapter 5 of the Deliverable D2.2.

To obtain postdate security we rely on the weak beacon and on signatures. The signer in our case queries

the beacon thus obtaining the pair $(\eta, \mathtt{tsl})$ where η represents the $\mathtt{tsl}$-th output of $\mathcal{B}^{\mathsf{w}}$ (which is also the most recent) and sign the message together with with η. In order to obtain backdate-security, the signer inserts its signature, via a transaction to the blockchain. Now, the signature is only considered validly timed after it appears on the ledger's state and is posted within a predefined delay. Moreover, as we will prove, combining the above two ideas yields a signature with both backdate and postdate security.

5.1.3 Timed Zero-Knowledge PoK (TPoK) and Signature of Knowledge (TSoK)

TPoK. In this section we apply the same methodology used for timed signatures in the previous section to define analogously timed versions of non-interactive zero-knowledge proofs of knowledge. The basis for our approach is the standard UC Non-Interactive Zero-Knowledge functionality proposed $\mathcal{F}_{\mathsf{NIZK}}$ in [GOS12b]. Roughly, $\mathcal{F}_{\mathsf{NIZK}}$ considers two parties, a prover and a verifier. The prover provides as input to $\mathcal{F}_{\mathsf{NIZK}}$ an $\mathcal{NP}$-statement x. The functionality checks whether $(x, w) \in \mathcal{R}$ (where $\mathcal{R}$ is an $\mathcal{NP}$-relation) and if the check is successful then $\mathcal{F}_{\mathsf{NIZK}}$ stores (x, π) and sends a proof π to the prover. The verifier can query the functionality with a couple (x, π), and if $\mathcal{F}_{\mathsf{NIZK}}$ stores the couple (x, π) sends 1 to the verifier, 0 otherwise.

We extend $\mathcal{F}_{\mathsf{NIZK}}$ to consider different levels of timed-security, in the same way as we have done for signatures: A proof π generated with respect to an $\mathcal{NP}$-statement is equipped with a time-mark $\mathtt{tsl}$ that gives some information about when π was computed. We refer to this notion of NIZK as TPoK and, also in this case, we consider three categories of security: *backdate*, *postdate* and *timed security*. The formalization of these notions is given by means of the UC-functionalities $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$, with $\mathtt{t} = \text{“}-\text{”}, \text{“}+\text{”}, \text{“}\pm\text{”}$. $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$ is formally described in the Chapter 5 of the Deliverable 2.2. In this setting, intuitively, a prover can send to the functionality a couple (x, w), and if $(x, w) \in \mathcal{R}$ then $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$ returns a couple $(\pi, \mathtt{tsl})$, where $\mathtt{tsl}$ is a time-mark. A verifier that queries $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$ with a couple $(\pi, \mathtt{tsl})$ and gets 1 as the answer from the functionality has the guarantees that: 1) the prover knew the witness for the $\mathcal{NP}$-statement x (like in case of $\mathcal{F}_{\mathsf{NIZK}}$) and 2) the proof π was generated (using the witness) in some moment specified by $\mathtt{tsl}$. We also provide three instantiations, one for each of the three security notions mentioned above. That is, we show a protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$ that UC-realize $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathtt{t}}$ for all $\mathtt{t} \in \{-, +, \pm\}$. $\Pi_{\mathsf{TPoK}}^{\mathsf{w},-}$ is similar to $\Pi_{\sigma}^{\mathsf{w},-}$, indeed the prover of $\Pi_{\mathsf{TPoK}}^{\mathsf{w},-}$, on input $(x, w) \in \mathcal{R}$, just needs to compute a NIZK proof (e.g. computed using $\mathcal{F}_{\mathsf{NIZK}}$) and stores it into the ledger. $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ instead needs to use $\mathcal{B}^{\mathsf{w}}$. $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ follows the *commit-and-prove* paradigm in which the prover commits to the witness w for the $\mathcal{NP}$-statement x to be proven, and then proves to the verifier that the committed message corresponds to a valid witness for x. In our protocol we want to associate some time-stamp to the proof generated by the prover, so we slightly modify the above approach as follows. The prover obtains the pair $(\eta, \mathtt{tsl})$ by invoking the weak beacon $\mathcal{B}^{\mathsf{w}}$ with the current round ρ, where η represents the $\mathtt{tsl}$-th output of $\mathcal{B}^{\mathsf{w}}$ (which is also the most recent).

Then the prover computes a commitment $\mathtt{com}$ of $w||\eta$ and proves to the verifier that $\mathtt{com}$ contains a witness for x concatenated with η. The verifier, upon receiving the proof computed by the prover accepts it if and only if the following two conditions hold: 1) value η has been output by $\mathcal{B}^{\mathsf{w}}$ in some round τ; 2) the NIZK proof given by the prover is accepting. Since the NIZK that we use is a PoK and we assume that a malicious prover cannot predict the output of the weak beacon $\mathcal{B}^{\mathsf{w}}$ more than $\delta = \mathtt{MaxRound} \cdot (\mathtt{WindowSize} + \ell - \mu)$ rounds in advance then the verifier has the guarantee that the proof has been computed (and that the witness w was known by the prover) in some moment *subsequent* to $\tau - \delta$. The protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ instead internally runs $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ and stores the proof on the ledger, and this is sufficient to provide timed security.

TSoK. The work of Chase *et al.* [CL06] introduces the notion of *signature of knowledge (SoK)*. A signature of knowledge schemes allows to issue signatures on behalf of any $\mathcal{NP}$-statement. That is, receiving a valid signature for a message m with respect to an $\mathcal{NP}$-statement x means that the signer of m knew the witness w for the $\mathcal{NP}$-statement x. Exactly in the same spirit of signature and NIZK, we define the notions of backdate, postdate and timed SoK. That is, we define the UC-functionalities $\mathcal{F}_{\mathsf{TSoK}}^{\mathsf{w},\mathtt{t}}$ with $\mathtt{t} \in \{-, +, \pm\}$. Those functionalities are analogous to the functionalities for timed signatures and NIZKs we have just discussed. It should be easy to see that postdate, backdate and timed secure NIZK implies respectively postdate, backdate and timed secure TSoK.

Indeed we observe in the $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathsf{t}}$-hybrid model it is possible to obtain a protocol that UC-realizes $\mathcal{F}_{\mathsf{TSoK}}^{\mathsf{w},\mathsf{t}}$. The approach is to construct a commitment of a witness concatenated with the message that we want to sign and then run $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathsf{t}}$ to prove that the commitment actually contains the concatenation of the witness for x and the message m.

5.2 Weak Block Unpredictability (WBU)

A delicate point about the ledger from [BMTZ17, BGK$^+$18] is the way it enforces the chain quality property from [GKL15]. Recall that this property requires that in every sequence of ℓ blocks put into the state, at least μ of them have to be associated with honest leaders. The ledger enforces this by the simulator declaring in a special field—corresponding to a coinbase transaction—the identity of the party who should be considered as having inserted each block; the extend-policy predicate will then ensure that the simulator has to declare blocks as created by honest parties with a sufficiently high frequency as above.

Our analysis—as well as the security analyses of the ledger [BMTZ17, BGK$^+$18] and the backbone abstraction of the protocol [GKL15, PSS17]—uses the assumption that the coinbase transaction of such *honest* blocks includes at least $\hat{\lambda}$ bits randomly chosen by an honest party[1]. One might be tempted to deduce that it is possible to extract (at least) $\hat{\lambda}$ bits of randomness from each sequence of ℓ blocks. However, this is not the case. Informally, the reason is that parties are in parallel working to extend the chain, and there is a chance that they might collide, giving the adversary the choice between the colliding blocks. And, although, one can use the existence of uniquely successful rounds—i.e., rounds in which only one honest party succeeds in solving the PoW puzzle—guaranteed to exist by the analysis of [GKL15], this is not sufficient: The problem is that the most recent part of the blockchain is not stable (it is not part of the common prefix) so the adversary can, in principle overwrite it, potentially using alternative postfixes (which can include blocks even by honest parties that have inconsistent view of the blockchain's head). This gives the adversary a bit more slackness in guessing the output of the beacon. Informally, the entropy of the honest block can be reduced by a factor that depends on the number of honest blocks proposed within a small window from the round in which the beacon emits its value. However, as we will argue below, this grinding might at most eliminate a few bits of entropy from the beacon.

Attempting to capture the above, we hit a shortcoming of the ledger from [BMTZ17]. The reason is that in the current definition of the ledger, there is no way for an honest party to insert some random value into a block's content, as the ledger allows its simulator to have full control of the contents of the blocks inserted into the state. Note that the extend policy algorithm (responsible for enforcing the chain quality and liveness) in the ledger functionality does not account for the above property. A way to rectify that would be to adjust the extend policy, but this would then mean changing the ledger in a non-transparent manner.

Instead, here we choose to take the following approach, which was also proposed in [BMTZ17] for explicitly capturing assumptions—in the case of [BMTZ17] it was used for capturing honest majority of computing power: We introduce an explicit wrapper that exactly captures the property that yields the above entropic argument. We refer to this wrapper as WBU-wrapper, and to the corresponding property that it enforces as weak beacon unpredictability, and denote it as $\mathcal{W}_{\mathrm{WBU}}$.

In a nutshell, the WBU-wrapper wraps the ledger functionality, i.e., takes control of all its interfaces, and acts as a relayer except for the following behavior: It might accept a special input from the simulator in any round (even multiple times per round). Once it does, it returns a random nonce N and records the pair (N, ρ), where ρ is the current round. Furthermore, for each block inserted into the state, it records the block along with the round in which this insertion occurred (note that the wrapper can easily detect insertions by reading the state through all miner's interfaces). If it observes that the simulator does not ask for a nonce for more than $\ell \cdot \texttt{MaxRound}$ rounds, or does not insert a block with its coinbase including a previously output nonce N within a δ-long time window from the creation of N, where $\delta = \texttt{MaxRound} \cdot \texttt{WindowSize} \cdot (\ell - \mu)$, then the wrapper halts. The formal definition of the weak block unpredictability wrapper is as follows.

[1] Formally, in [BMTZ17, BGK$^+$18] the ledger chooses the contents of the coinbase transactions of honest blocks, including the nonces and possible new keys/wallet-addresses, hence the simulator cannot predict them.

Definition 15 (Weak Block Unpredictability Wrapper: $\mathcal{W}_{\text{WBU}}$)**.** A $\mathcal{W}_{\text{WBU}}$ is a functionality-wrapper (that wraps $\mathcal{G}_{\texttt{ledger}}$) and operates as follows:

- Upon receiving (new_nonce) from the simulator it returns random fresh $N \in \{0,1\}^{\lambda}$ to the simulator, and records (N, ρ), where ρ is the current round ($\mathcal{W}_{\text{WBU}}$ can get this round by querying the clock.)
- For any block proposed by the simulator that makes it into the ledger's state, which is flagged (via the coinbase transaction, by the simulator) as originating from an honest party ($\mathcal{W}_{\text{WBU}}$ can detect this as discussed above). If this block does not contain some N previously recorded, then halt; otherwise, if (N, ρ') has been recorded and the current round index is $\rho > \rho' + \delta = \rho' + \texttt{MaxRound} \cdot \texttt{WindowSize} \cdot (\ell - \mu)$ then halt. In any other case simply relay messages between the wrapped functionality and the entities it is connected to (i.e., the simulator, the environment, and the global setups it has registered with.)

As an additional contribution of this work, which we believe might be of independent interest, we prove the following lemma which states that the (UC abstraction of the) Bitcoin backbone protocol from [BMTZ17] emulates the wrapped ledger $\mathcal{W}_{\text{WBU}}[\mathcal{G}_{\texttt{ledger}}]$, where, $\mathcal{G}_{\texttt{ledger}}$ is the ledger from [BMTZ17]. We prove that by showing that the blocks generated by the protocol satisfy the weak beacon unpredictability property. The lemma follows then directly by observing that the simulator of [BMTZ17] internally generates the coinbase for honest blocks by emulating the honest protocol.

Lemma 5. *The (UC version of the) Bitcoin backbone protocol* Π_{BB} *[BMTZ17] realizes* $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$.

Proof. Let κ be the security parameter of Π_{BB}, $\delta = \texttt{MaxRound} \cdot \texttt{WindowSize} \cdot (\ell - \mu)$ and $\lambda = \hat{\lambda} - \log(\kappa)$.

We start the proof by showing that in Π_{BB}, under the assumption that the honest parties include a random value of length $\hat{\lambda}$ in each coinbase transaction, no adversary can predict at round ρ, with probability greater than $2^{-\lambda}$, the nonce η' that is added by an honest party to a block that becomes part of state at round $\rho + \delta$. Let ρ_1 be the round in which an honest party p_i sees $\mathsf{st} = \mathsf{state}_{|p_i}$, then the block that extends st could either be a block generated by an honest party, or a block generated by a malicious party. Moreover, this new block has to be added to st after at most $\texttt{MaxRound} \cdot \texttt{WindowSize}$ rounds. In the case that the block that extends st is malicious, we cannot say much on the entropy since, without loss of generality, we assume that the content of the block is in full control of the adversary. Let us now consider the case in which the added block is honest. Let ρ_2 be the round in which the block becomes part of the ledger state. We note that all honest parties that generate candidate blocks[2] for state at round ρ_2 could see state already at round ρ_1 with $\mathcal{T} = |\rho_2 - \rho_1| \leq \texttt{MaxRound} \cdot \texttt{WindowSize}$. We now want to compute t, which represents the number of all the possible candidate blocks that can be seen by the adversary in the interval $[\rho_1, \rho_2]$. Note that there could be other candidate blocks for ledger states that are shorter than state, but those blocks cannot be used by the adversary anymore.

From [GKL15, Remark 3] we know that the number of blocks generated by the honest parties in an interval of size $\mathcal{T}$ is $t = pq(n-m)\mathcal{T}$, where n is to the total number of parties, m is the number of parties controlled by the adversary, p is the probability that an honest party generates a block and q is the upper bound on the number of queries that each party can make to the random oracle. Moreover, from the proof of [GKL15, Lemma 6] we know that $pq(n-m) \leq \frac{1}{2}$, therefore $t \leq \mathcal{T}/2 \leq \texttt{MaxRound} \cdot \texttt{WindowSize}/2$. Given that WindowSize and MaxRound are polynomially related to the security parameter of the ledger κ, we have that $t = \mathsf{poly}(\kappa)$.

From the chain quality we also know that after *at most* $\ell - \mu$ blocks, an honestly generated block has to be added to the ledger state. This means that if $\delta = \texttt{MaxRound} \cdot (\texttt{WindowSize} + \ell - \mu)$, then by round $\rho + \delta$ there is one block included in the ledger state that has min-entropy $\lambda = \hat{\lambda} - \log(\kappa)$, conditional on the view of the adversary at round ρ. We are now ready to show that Π_{BB} implements $\mathcal{W}_{\text{WBU}}[\mathcal{G}_{\texttt{ledger}}]$. We describe a corresponding polynomial-time simulator Sim. Let Sim_{BB} be the simulator for Π_{BB}. Sim acts as the ideal-functionality $\mathcal{G}_{\texttt{ledger}}$ for Sim_{BB} with the following difference. Whenever it is required to compute a nonce for the coinbase transaction, Sim queries $\mathcal{W}_{\text{WBU}}[\mathcal{G}_{\texttt{ledger}}]$ with new_nonce thus obtaining (N, ρ) and uses N as the nonce with $N \in \{0,1\}^{\hat{\lambda}}$.

□

[2] In this proof we call *candidate block* a block that could extend state.

5.3 The (Weak) Beacon functionality

5.3.1 Our weak beacon protocol

In this section, we propose a protocol that realizes the $\mathcal{B}^{\mathsf{w}}$ functionality in the $(\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}}), \mathcal{F}_{RO})$-hybrid model. We first recall some of the properties that $\mathcal{G}_{\mathtt{ledger}}$ (similarly $\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}})$) enjoys, and will be useful here.

1. The chain quality property of $\mathcal{G}_{\mathtt{ledger}}$ guarantees that any portion of state of length ℓ contains, at least, a portion of μ blocks originated by some honest parties. This means that the remaining $\ell - \mu$ blocks might be chosen by the adversary and the contents of these blocks are under full control of the adversary.

2. Since honest blocks include at least λ bits of entropy, the output of a hash function modeled as a RO with security parameter λ on input an honest block represents a uniform random value in $\{0,1\}^{\lambda}$.

At a high level, our beacon protocol works as follows. A party that wants to compute the beacon's output reads state from $\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}})$ and outputs the hash of the latest $\ell - \mu + 1$ blocks of state. At first glance, as any chunk of $\ell - \mu + 1$ blocks of state contains (at least) an honestly generated block, the output of the beacon is an unpredictable random value. However, this is not the case. The first observation is that, using the technique described above, an adversary can predict the next $\ell - \mu$ outputs of the beacon in advance. In particular, the adversary first allows a sequence of μ honestly generated blocks to be added to the chain and then it inserts its own $\ell - \mu$ pre-computed adversarial blocks after those μ blocks. But, the *prediction power* of the adversary is not limited to $\ell - \mu$ blocks. We recall that the view that an honest party has of the ledger state could differ of at most $\mathtt{WindowSize}$ blocks. Therefore, in the worst case, the adversary sees $\mathtt{WindowSize}$ blocks in advance with respect to an honest party, thus giving an additional prediction power to him. In conclusion we can claim that, given a ledger $\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}})$ with chain quality parameters (μ, ℓ) and window size $\mathtt{WindowSize}$, it is possible to construct a weak beacon $\mathcal{B}^{\mathsf{w}}$ in which an adversary can predict, with respect to an honest party, the next $\Delta = \ell - \mu + \mathtt{WindowSize}$ outputs. This means that an adversary, in the worst case, can predict the outcome of the beacon $\delta = \mathtt{MaxRound} \cdot \Delta$ rounds in advance with respect to an honest party, where $\mathtt{MaxRound}$ represents the liveness parameter of $\mathcal{G}_{\mathtt{ledger}}$. The only thing left to argue is how the output of the beacon is distributed. In the above scenario, not only the adversary can predict the next $\ell - \mu$ outputs, but can also bias those outputs since he can decide to extend state with any sequence of $\ell - \mu$ blocks. Since we model the hash function as a random oracle, it is easy to see that the bias of the output of the beacon depends on the randomness inside the honest blocks and on the hashing power of the adversary. Indeed, a powerful adversary can always decide what the next $\ell - \mu$ of state will be in the worst case. In this work we denote with $\mathtt{MaxSize}$ the maximum number of queries that the adversary can ask the RO. We observe that our instantiation of the weak beacon only needs to read information from the ledger. In Fig. 5.1 we provide a formal construction of the weak beacon protocol Π^{w} that UC-realizes $\mathcal{B}^{\mathsf{w}}$ with $\mathsf{w} = ((\mu, \ell), \mathtt{MaxRound}, \mathtt{WindowSize}, \mathtt{MaxSize})$. The steps described in Fig. 5.1 follow the description given here with the exception that all the parties invoke the procedure update_time_table() every time that an input is received (see Fig. 5.2). This procedure helps a party to keep track of the size of the ledger (i.e. the size of state) at any round.

Theorem 7. *Let $\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}})$ be the wrapper functionality for $\mathcal{G}_{\mathtt{ledger}}$ defined in Def. 15 parameterized by $((\mu, \ell), \mathtt{MaxRound}, \mathtt{WindowSize})$, then protocol Π^{w} described in Fig. 5.1 securely realizes $\mathcal{B}^{\mathsf{w}}$ in the $(\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}}), \mathcal{G}_{\mathtt{clock}}, \mathcal{F}_{RO})$-hybrid model with $\mathsf{w} = ((\mu, \ell), \mathtt{MaxRound}, \mathtt{WindowSize}, \mathtt{MaxSize})$ where $\mathtt{MaxSize} = \mathsf{poly}(\lambda)$.*

Proof. Let $\mathcal{A}$ be an arbitrary polynomial time adversary. We will describe a corresponding polynomial time ideal process adversary Sim such that no non-uniform polynomial time environment can distinguish whether Π^{w} is running in the $(\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\mathtt{ledger}}), \mathcal{G}_{\mathtt{clock}}, \mathcal{F}_{RO})$-hybrid model with parties $p_1, \ldots, p_n$ and adversary $\mathcal{A}$ or the ideal process is running with $\mathcal{B}^{\mathsf{w}}$, Sim and dummy parties $\tilde{p}_1, \ldots, \tilde{p}_n$. Sim starts by invoking a copy of $\mathcal{A}$. It will run a simulated interaction of $\mathcal{A}$, the parties and the environment. In particular, whenever the simulated $\mathcal{A}$ communicates with the environment, Sim just passes this information along. And whenever $\mathcal{A}$ corrupts a party p_i, Sim corrupts the corresponding dummy party $\tilde{p}_i$.

We assume that a set of parties $\mathcal{P}$ are registered to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$, $\mathcal{G}_{\texttt{clock}}$ and $\mathcal{F}_{RO}$. Every time that a party $p_i \in \mathcal{P}$ receives an input it invokes the procedure update_time_table(). Let $\texttt{cq} \leftarrow \ell - \mu + 1$. The party p_i on input $(\leftarrow\!\$, \tau, sid)$ proceeds as follows.

1. Send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ and wait for an answer.
2. Upon receiving $(\texttt{READ}, sid, \mathsf{state})$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ set $\texttt{tsl} \leftarrow \mathcal{T}_{p_i}^{\texttt{local}}(\tau)$ and send $(\mathsf{Eval}, sid, \texttt{tsl}||\mathsf{state}_{|\texttt{tsl}-\texttt{cq},\texttt{tsl}})$ to $\mathcal{F}_{RO}$.
3. Upon receiving the answer $(\mathsf{Eval}, sid, \eta)$ from $\mathcal{F}_{RO}$ output $(\leftarrow\!\$, sid, \texttt{tsl}, \eta)$.

Figure 5.1: The weak beacon protocol in $(\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}}), \mathcal{G}_{\texttt{clock}}, \mathcal{F}_{RO})$-hybrid model.

update_time_table()
When the procedure is invoked by $p_i \in \mathcal{P}$ the query $(\texttt{CLOCK-READ}, sid_C)$ is sent to $\mathcal{G}_{\texttt{clock}}$. Upon receiving the answer $(\texttt{CLOCK-READ}, sid_C, R)$ from $\mathcal{G}_{\texttt{clock}}$, send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Upon receiving receiving $(\texttt{READ}, sid, \mathsf{state})$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ set $\mathcal{T}_{p_i}^{\texttt{local}}(R) := |\mathsf{state}|$.

Figure 5.2: The procedure update_time_table().

In the following description of Sim, we denote with head the current size of the state state of $\mathcal{G}_{\texttt{ledger}}$. The behavior of Sim can be summarized as follows.

1. Whenever $\mathcal{A}$ sends $(\mathsf{Eval}, sid, i||x)$ to $\mathcal{F}_{RO}$, Sim does the following.
   - If $i > \texttt{head} + \ell - \mu$, then pick a random value $\rho \in \{0,1\}^\lambda$ and instruct $\mathcal{F}_{RO}$ to reply with $(i||x, \rho)$ any time that $(\mathsf{Eval}, sid, i||x)$ is received.
   - If $\texttt{head} < i \leq \texttt{head}+\ell-\mu$, then send $(\texttt{READ\_SETS}, sid)$ to $\mathcal{B}^{\mathsf{w}}$. Upon receiving $(\texttt{READ\_SETS}, sid, \mathcal{S}_1, \ldots, \mathcal{S}_{\ell-\mu})$, take randomly a new element η from $\mathcal{S}_{i-\texttt{head}} - \tilde{\mathcal{S}}_{i-\texttt{head}}$, $\tilde{\mathcal{S}}_{i-\texttt{head}} \leftarrow \tilde{\mathcal{S}}_{i-\texttt{head}} \cup \{\eta\}$ and instruct $\mathcal{F}_{RO}$ to reply with η on the query $(\mathsf{Eval}, sid, i||x)$.
2. Any time that state is extended with a new block Block, Sim checks if Block is generated honestly by reading the value hflag. Let $\texttt{cq} \leftarrow \ell - \mu + 1$. Sim computes $x \leftarrow \mathsf{state}_{|\texttt{head},\texttt{head}-\texttt{cq}}$ and does the following.
   - If $\texttt{hflag} = 1$ then send $I = (\texttt{SET\_RANDOM}, sid)$ to $\mathcal{B}^{\mathsf{w}}$. If $\mathcal{B}^{\mathsf{w}}$ replies with $(\texttt{OK}, sid, \eta)$, then instruct $\mathcal{F}_{RO}$ to reply with η on the query $(\mathsf{Eval}, sid, \texttt{head}||x)$ and set $\tilde{\mathcal{S}}_1 = \emptyset, \ldots, \tilde{\mathcal{S}}_{\ell-\mu} = \emptyset$
   - if $\texttt{hflag} = 0$ check if $\mathcal{F}_{RO}$ has been queried with $(\mathsf{Eval}, sid, \texttt{head}||x)$. If it has, then send $(\texttt{SET}, sid, x)$ to $\mathcal{B}^{\mathsf{w}}$ else send $I = (\texttt{SET\_RANDOM}, sid)$. We observe that if $(\mathsf{Eval}, sid, \texttt{head}||x)$ has been queried before by the adversary, then $\eta \in \mathcal{S}_{(\texttt{head} \mod \ell-\mu)}$.
3. At any round, Sim updates $\mathcal{T}$ according to the slackness values of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. That is, Sim reads the values $\texttt{pt}_{i_1}, \ldots, \texttt{pt}_{i_n}$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$, defines a time-table $\mathcal{T}'$ that extends the previous one ($\mathcal{T}$) by setting $\mathcal{T}'[R, p_{i_1}] = \texttt{pt}_{i_j}$ for all the honest parties p_{i_j}, and set $\mathcal{T}'[\tau, p_{i_k}] = |\mathsf{state}|$ for all the corrupted parties p_{i_k}. Then Sim sends $(\texttt{SET-DELAYS}, sid, \mathcal{T}')$ to $\mathcal{B}^{\mathsf{w}}$.

We now observe that Sim in step 3 keeps consistent the view that each party has of state with the view of $\mathcal{H}$. That is, each party that in the ideal model can see the i-th value of $\mathcal{H}$ can see also the i-th block of state in the real world. This is done in step 3 by updating the time-table $\mathcal{T}$ consistently with the slackness of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Moreover, any time that state is extended with a new block, $\mathcal{H}$ is extended as well. As showed in step 2, $\mathcal{H}$ is

extended via adding a random value decided by the functionality if state was extended honestly, or by using a value taken from the set $\mathcal{S}_i$ with $i = \texttt{head} \mod \ell - \mu$. The crucial observation here is that, because honest blocks include at least λ bits of fresh entropy, if state is extended with an honest block then the output of the RO on input $(\mathsf{Eval}, sid, \mathsf{state}_{(\texttt{head}-\ell+\mu+1,\texttt{head})})$ is a uniform in $\{0,1\}^\lambda$. Moreover, since any consecutive blocks of state contain, at least, μ honestly generated blocks, this guarantees that the adversary can only predict the next $\ell - \mu$ outputs. In step 1 Sim takes care of the latter aspect by keeping consistent the content of the sets $\mathcal{S}_1, \ldots, \mathcal{S}_{\ell-\mu}$ with the queries made by the real world adversary to the RO. In more details, once that Sim gets the sets $\mathcal{S}_1, \ldots, \mathcal{S}_{\ell-\mu}$ from $\mathcal{B}^{\mathsf{w}}$ then he instructs the RO to reply to the query $(\mathsf{Eval}, sid, j||x)$ with η, where η is randomly chosen from $\mathcal{S}_{j \mod \ell-\mu}$. We recall that these sets have size MaxSize where MaxSize denotes an upper bound on the number of queries that can be made by the adversary to the RO. We also observe that, in order to reply with different values of $\mathcal{S}_{j \mod \ell-\mu}$ to the queries $(\mathsf{Eval}, sid, j||x)$, $(\mathsf{Eval}, sid, j||x')$ with $x \neq x'$, Sim keep track of all the values of $\mathcal{S}_{j \mod \ell-\mu}$ that have been already used to program the RO in a special set $\tilde{\mathcal{S}}_{j \mod \ell-\mu}$.

□

5.3.2 Discussion on alternative constructions

By looking at some real blockchains such as the Bitcoin blockchain, one would be tempted to implement $\mathcal{B}^{\mathsf{w}}$ by hashing only the last (stable) block of the hash chain. This might actually be a more efficient way to implement our weak-beacon functionality than the one in this work. However, it would not be a *generic* approach and is suitable for a certain type of blockchains, e.g., Bitcoin. In particular, the alternative approach would require some properties of blockchains not captured by $\mathcal{G}_{\texttt{ledger}}$. Indeed, for the case of Bitcoin we have that the blocks are organized in a hash-chain—where the hash function behaves as a random oracle—and those aspects are not exported to the UC Ledger. Our construction works for arbitrary blockchains, as it uses the UC ideal ledger functionality by Badertscher et al. in a black-box (ideal) manner wrapped with $\mathcal{W}_{\text{WBU}}$. We use $\mathcal{W}_{\text{WBU}}$ to only capture the fact that honest blocks (which appear frequently according to chain quality) have sufficient entropy, which we prove is the case using only the basic properties of the Backbone protocol; for adversarial blocks the Ledger offers no unpredictability guarantees. Hence, the alternative construction cannot work unless we make extra assumptions on the structure of the blockchain and hence on the entropy of the maliciously generated blocks. We remark that even though one might be able to prove that Bitcoin's output does have such extra properties—sufficient for the above alternative construction—the resulting statement would not be stronger: one would still need to rely on the unpredictability of the next honest block and on chain quality; hence it would at most give a slightly more efficient solution (hashing one block instead of $\ell - \mu$) at the cost of a more involved analysis with extra assumptions.

5.4 Timed Signatures

We provide a scheme $\Pi_\sigma^{\mathsf{w},\mathtt{t}}$ that UC-realizes the functionality $\mathcal{F}_\sigma^{\mathsf{w},\mathtt{t}}$ for $\mathtt{t} = \text{“}-\text{”}, \text{“}+\text{”}, \text{“}\pm\text{”}$. The backdate secure TSign scheme $\Pi_\sigma^{\mathsf{w},-}$ showed in Fig. 5.3 realizes $\mathcal{F}_\sigma^{\mathsf{w},-}$ in the $(\mathcal{F}_{\texttt{SIGN}}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model. In this, informally, the signer signs a message m using a standard signature scheme, creates a transaction that contains the signature of the message and then asks the ledger to store the transaction permanently into state. Intuitively, the construction is secure as of the security of the signature scheme and because the history of the ledger cannot be changed. Also, $\Pi_\sigma^{\mathsf{w},-}$ realizes $\mathcal{F}_\sigma^{\mathsf{w},-}$ with $\mathsf{w} = (\perp, \texttt{MaxRound}, \texttt{WindowSize}, \texttt{waitingTime})$ where MaxRound, WindowSize and waitingTime are the parameters of $\mathcal{G}_{\texttt{ledger}}$. In this construction, as for the weak beacon protocol, every honest party p_i maintains a table $\mathcal{T}_{p_i}^{\texttt{local}}$ that is updated on any input received by p_i according to the procedure update_time_table(). The aim of $\mathcal{T}_{p_i}^{\texttt{local}}$ is to associate each round of $\mathcal{G}_{\texttt{clock}}$ to the size of the state of $\mathcal{G}_{\texttt{ledger}}$. The postdate secure TSign scheme $\Pi_\sigma^{\mathsf{w},+}$, showed in the Fig. 5.4, realizes $\mathcal{F}_\sigma^{\mathsf{w},+}$ in the $(\mathcal{F}_{\texttt{SIGN}}, \mathcal{B}^{\mathsf{w}}, \mathcal{G}_{\texttt{clock}})$-hybrid model. In this protocol the signer, on input a message m, first invokes the weak beacon $\mathcal{B}^{\mathsf{w}}$ thus obtaining the most recent output (in his view) η and then signs $m||\eta$ using a standard signature

scheme. Intuitively, this scheme is secure due to the unforgeability of the signature scheme and because an adversary can, in the worst case, predict only the future Δ outputs of the beacon. More precisely, $\Pi_{\sigma}^{\mathsf{w},+}$ realizes $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ with $\mathsf{w} = (\Delta = (\ell, \mu), \texttt{MaxRound}, \texttt{WindowSize}, \perp)$ where MaxRound, WindowSize are the parameters of $\mathcal{B}^{\mathsf{w}}$.

The timed secure TSign scheme $\Pi_{\sigma}^{\mathsf{w},\pm}$ showed in Fig. 5.5 realizes $\mathcal{F}_{\sigma}^{\mathsf{w},\pm}$ in the $(\mathcal{F}_{\sigma}^{\mathsf{w},+}, \mathcal{B}^{\mathsf{w}}, \mathcal{G}_{\texttt{clock}})$-hybrid model. In the construction we promote the postdate security of $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ to timed security by simply storing the signature obtained via $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ into the ledger. The security of this scheme follows immediately from the postdate security of $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ and from the immutability of $\mathcal{G}_{\texttt{ledger}}$.

We assume that the parties $\mathcal{P}$ are registered to $\mathcal{G}_{\texttt{ledger}}$, $\mathcal{G}_{\texttt{clock}}$ and $\mathcal{F}_{\texttt{SIGN}}$. Each party $p_i \in \mathcal{P}$ manages a local time table $\mathcal{T}_{p_i}^{\texttt{local}}$ that is updated any time that p_i receives an input by invoking the procedure update_time_table().
Initialization. The signer $S \in \mathcal{P}$ sends $(\texttt{KEY\_GEN}, sid)$ to $\mathcal{F}_{\texttt{SIGN}}$ thus obtaining $(\texttt{VERIFICATION\_KEY}, sid, v)$.
Signature. The signer $S \in \mathcal{P}$ on input $(\texttt{TIMED\_SIGN}, sid, -, m, \tau_{\texttt{req}})$ executes the following steps.

1. Send $(\texttt{SIGN}, sid, m)$ to $\mathcal{F}_{\texttt{SIGN}}$ and upon receiving the answer $(\texttt{SIGNATURE}, sid, m, \sigma)$, create a transaction $\texttt{tx} := (m, \sigma)$ and send $(\texttt{SUBMIT}, sid, \texttt{tx})$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$.
2. Wait until tx is added to the state of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Let $\texttt{tsl}_{\texttt{back}}$ be the block of state that contains tx, output $(\texttt{TIMED\_SIGNATURE}, sid, -, (\texttt{tsl}_{\texttt{back}}, m, \sigma, \perp))$.

Verification. The party $p_i \in \mathcal{P}$ on input $(\texttt{TIMED\_VERIFY}, sid, -, (\texttt{tsl}_{\texttt{back}}, m, \sigma, v', \perp))$ proceeds as follows.

1. Send $(\mathsf{Vf}, sid, m, \sigma, v')$ to $\mathcal{F}_{\texttt{SIGN}}$.
2. Upon receiving $(\texttt{VERIFIED}, sid, m, b)$ from $\mathcal{F}_{\texttt{SIGN}}$, if $b = 1$ then send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ else output $(\texttt{TIMED\_VERIFIED}, sid, -, (\perp, m, 0, \perp))$.
3. Upon receiving the answer $(\texttt{READ}, sid, \mathsf{state})$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ check if the transaction $\texttt{tx} = (m, \sigma)$ is stored in the $\texttt{tsl}_{\texttt{back}}$-th block of state. If it is, then find the smallest τ_{back} such that $\mathcal{T}_{p_i}^{\texttt{local}}[\tau_{back}] = \texttt{tsl}_{\texttt{back}}$ and output $(\texttt{TIMED\_VERIFIED}, sid, -, (\tau_{back}, m, 1, \perp))$ otherwise output $(\texttt{TIMED\_VERIFIED}, sid, -, (\perp, m, 0, \perp))$.

Figure 5.3: The protocol $\Pi_{\sigma}^{\mathsf{w},-}$ in the $(\mathcal{F}_{\texttt{SIGN}}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model.

Theorem 8. *The protocol $\Pi_{\sigma}^{\mathsf{w},-}$ described in Fig. 5.3 realizes with perfect security $\mathcal{F}_{\sigma}^{\mathsf{w},-}$ in the $(\mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}}, \mathcal{F}_{\texttt{SIGN}})$-hybrid model where $\mathcal{G}_{\texttt{ledger}}$ is parameterized by $((\mu, \ell), \texttt{MaxRound}, \texttt{WindowSize}, \texttt{waitingTime})$.*

Proof. Let $\mathcal{A}$ be an arbitrary polynomial time adversary. We will describe a corresponding polynomial time ideal process adversary Sim such that no non-uniform polynomial time environment can distinguish whether $\Pi_{\sigma}^{\mathsf{w},-}$ is running in the $(\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}}), \mathcal{G}_{\texttt{clock}}, \mathcal{F}_{\texttt{SIGN}})$-hybrid model with parties $p_1, \ldots, p_n$ and adversary $\mathcal{A}$ or the ideal process is running with $\mathcal{F}_{\sigma}^{\mathsf{w},-}$, Sim and dummy parties $\tilde{p}_1, \ldots, \tilde{p}_n$. Sim starts by invoking a copy of $\mathcal{A}$. It will run a simulated interaction of $\mathcal{A}$, the parties and the environment. In particular, whenever the simulated $\mathcal{A}$ communicates with the environment, Sim just passes this information along. And whenever $\mathcal{A}$ corrupts a party p_i, Sim corrupts the corresponding dummy party $\tilde{p}_i$. We summarize the behavior of Sim as follows.

- If $(\texttt{KEY\_GEN}, sid)$ is received then forward it to $\mathcal{F}_{\texttt{SIGN}}$ upon receiving the answer $(\texttt{VERIFICATION\_KEY}, sid, v)$ sent it back to $\mathcal{F}_{\sigma}^{\mathsf{w},-}$.
- If the input $(\texttt{TIMED\_SIGN}, sid, -, m, \tilde{S}, \tau_{\texttt{req}})$ is received it means that a dummy party $\tilde{S}$ has received the input $(\texttt{TIMED\_SIGN}, sid, -, m, \tau_{\texttt{req}})$. Therefore, send $(\texttt{SIGN}, sid, m)$ to $\mathcal{F}_{\texttt{SIGN}}$, and upon receiving $(\texttt{SIGNATURE}, sid, m, \sigma)$ generate the transaction $\texttt{tx} \leftarrow (m, \sigma)$ and send tx to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. When tx is added to the state state of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$, take the index of the block $\texttt{tsl}_{\texttt{back}}$ that contains tx and send $(\texttt{TIMED\_SIGNATURE}, sid, -, m, (\texttt{tsl}_{\texttt{back}}, \sigma, v, \perp))$ to $\mathcal{F}_{\sigma}^{\mathsf{w},-}$.

- If the input $(\texttt{TIMED\_VERIFIED}, sid, -, (\texttt{tsl}_{\texttt{back}}, m, \sigma, v', \bot))$ is received it means that a dummy party $\tilde{p}_i$ has received the input $(\texttt{TIMED\_VERIFY}, sid, -, (\texttt{tsl}_{\texttt{back}}, m, \sigma, v', \bot))$. Therefore Send $(\mathsf{Vf}, sid, m, \sigma, v')$ to $\mathcal{F}_{\texttt{SIGN}}$ and upon receiving the answer $(\texttt{VERIFIED}, sid, m, \phi)$ send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Upon receiving the answer from $(\texttt{READ}, sid, \mathsf{state})$ check if the block in the $\texttt{tsl}_{\texttt{back}}$-th position of state contains the transaction $\texttt{tx} = (m, \sigma)$. If it is not, then $\phi \leftarrow 0$. Send $(\texttt{TIMED\_VERIFIED}, sid, m, \phi)$ to $\mathcal{F}_{\sigma}^{\mathsf{w},-}$.
- At any round Sim updates $\mathcal{T}$ according to the slackness values of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. That is, Sim reads the values $\texttt{pt}_{i_1}, \ldots, \texttt{pt}_{i_n}$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$, defines a time-table $\mathcal{T}'$ that extends the previous one ($\mathcal{T}$) by setting $\mathcal{T}'[R, p_{i_1}] = \texttt{pt}_{i_j}$ for all the honest parties p_{i_j}, and set $\mathcal{T}'[\tau, p_{i_k}] = |\mathsf{state}|$ for all the corrupted parties p_{i_k}. Then Sim sends $(\texttt{SET-DELAYS}, sid, \mathcal{T}')$ to $\mathcal{F}_{\sigma}^{\mathsf{w},-}$.

We recall that the maximum number of blocks after that an honest transaction is added to the state of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ is $\texttt{waitingTime}$. Therefore, if the environment can distinguish between the ideal and the real execution it means that the security of either $\mathcal{F}_{\texttt{SIGN}}$ or $\mathcal{G}_{\texttt{ledger}}$ has been compromised.

□

We assume that the parties $\mathcal{P}$ are registered to $\mathcal{F}_{\texttt{SIGN}}$, $\mathcal{B}^{\mathsf{w}}$ and $\mathcal{G}_{\texttt{clock}}$.

Initialization. The signer $S \in \mathcal{P}$ sends $(\texttt{KEY\_GEN}, sid)$ to $\mathcal{F}_{\texttt{SIGN}}$ thus obtaining $(\texttt{VERIFICATION\_KEY}, sid, v)$.

Signature. The signer $S \in \mathcal{P}$ on input $(\texttt{TIMED\_SIGN}, sid, +, m, \tau_{\texttt{req}})$ executes the following steps.

- Send $(\leftarrow\!\$, \tau, sid)$ to $\mathcal{B}^{\mathsf{w}}$ and upon receiving $(\leftarrow\!\$, sid, \texttt{tsl}_{\texttt{post}}, \eta)$ send $(\texttt{SIGN}, sid, \texttt{tsl}_{\texttt{post}}||m||\eta)$ to $\mathcal{F}_{\texttt{SIGN}}$.
- Upon receiving $(\texttt{SIGNATURE}, sid, \texttt{tsl}_{\texttt{post}}||m||\eta, \sigma)$, $\sigma_t \leftarrow (\sigma, \eta)$ and output $(\texttt{TIMED\_SIGNATURE}, sid, +, (\bot, m, \sigma_t, \texttt{tsl}_{\texttt{post}}))$.

Verification. The party $p_i \in \mathcal{P}$ on input $(\texttt{TIMED\_VERIFY}, sid, +, (\bot, m, \sigma_t, v', \texttt{tsl}_{\texttt{post}}))$ parses σ_t as (σ, η) and executes the following steps.

- Parses σ_t as (σ, η) and query $\mathcal{B}^{\mathsf{w}}$ to check when (and if) the value η was issued by $\mathcal{B}^{\mathsf{w}}$. If η has never been issued by $\mathcal{B}^{\mathsf{w}}$ then output $(\texttt{TIMED\_VERIFIED}, sid, +, (\bot, m, 0, \bot))$. Else, let τ_{post} be the round in which η has been issued, send $(\mathsf{Vf}, sid, \texttt{tsl}_{\texttt{post}}||m||\eta, \sigma, v')$ to $\mathcal{F}_{\texttt{SIGN}}$.
- Upon receiving $(\texttt{VERIFICATION}, sid, m, b)$ from $\mathcal{F}_{\texttt{SIGN}}$, if $b = 1$ then output $(\texttt{TIMED\_VERIFIED}, sid, +, (\bot, m, 1, \tau_{post}))$ else output $(\texttt{TIMED\_VERIFIED}, sid, +, (\bot, m, 0, \bot))$

Figure 5.4: The protocol $\Pi_{\sigma}^{\mathsf{w},+}$ in the $(\mathcal{F}_{\texttt{SIGN}}, \mathcal{B}^{\mathsf{w}}, \mathcal{G}_{\texttt{clock}})$-hybrid model.

Theorem 9. *Let $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ be the functionality parameterized by $\mathsf{w} = (\Delta, \texttt{MaxRound}, \texttt{WindowSize}, \bot)$, then the protocol $\Pi_{\sigma}^{\mathsf{w},+}$ described in Fig. 5.4 realizes with perfect security $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ in the $(\mathcal{B}^{\mathsf{w}'}, \mathcal{G}_{\texttt{clock}}, \mathcal{F}_{\texttt{SIGN}})$-hybrid model where $\mathcal{B}^{\mathsf{w}'}$ is parameterized by $\mathsf{w}' = ((\mu, \ell), \texttt{MaxRound}, \texttt{WindowSize}, \texttt{MaxSize})$ with $\Delta = \ell - \mu$.*

Proof. Following the approach proposed in the proof of Theorem 8, we summarize the behavior of Sim as follows.

- If $(\texttt{KEY\_GEN}, sid)$ is received then forward it to $\mathcal{F}_{\texttt{SIGN}}$ upon receiving the answer $(\texttt{VERIFICATION\_KEY}, sid, v)$ sent it back to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$.
- If the input $(\texttt{TIMED\_SIGN}, sid, +, m, \tilde{S}, \tau_{\texttt{req}})$ is received it means that a dummy party $\tilde{S}$ has received the input $(\texttt{TIMED\_SIGN}, sid, +, m, \tau_{\texttt{req}})$. Therefore, send $(\leftarrow\!\$, sid, \tau_{\texttt{req}})$ to $\mathcal{B}^{\mathsf{w}}$, and upon receiving $(\leftarrow\!\$, \eta)$ send $(\texttt{SIGN}, sid, \texttt{tsl}_{\texttt{post}}||m||\eta)$ to $\mathcal{F}_{\texttt{SIGN}}$. Upon receiving $(\texttt{SIGNATURE}, sid, m', \sigma)$, define $\sigma_t \leftarrow (\sigma, \eta)$ send $(\texttt{TIMED\_SIGNATURE}, sid, +, m, (\bot, \sigma_t, v, \texttt{tsl}_{\texttt{post}}))$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$.

- If the input $(\texttt{TIMED\_VERIFIED}, sid, +, (\perp, m, \sigma, v', \texttt{tsl}_{\texttt{post}}))$ is received it means that a dummy party $\tilde{p}_i$ has received the input $(\texttt{TIMED\_VERIFY}, sid, +, (\perp, m, \sigma_t, v', \texttt{tsl}_{\texttt{post}}))$. Therefore parse σ_t as (σ, η) and send $(\mathsf{Vf}, sid, \texttt{tsl}_{\texttt{post}}||m||\eta, \sigma, v')$ to $\mathcal{F}_{\texttt{SIGN}}$ and upon receiving the answer $(\texttt{VERIFIED}, sid, m, \phi)$ query the $\mathcal{B}^{\mathsf{w}}$ to check if the value η is the $\texttt{tsl}_{\texttt{post}}$-th output of $\mathcal{B}^{\mathsf{w}}$. If it is not then set $\phi \leftarrow 0$ and send $(\texttt{TIMED\_VERIFIED}, sid, m, \phi)$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$.
- At any round Sim updates $\mathcal{T}$ according to the time table that is managed by $\mathcal{B}^{\mathsf{w}}$. That is, let $\mathcal{T}'$ be the time table of $\mathcal{B}^{\mathsf{w}}$, then Sim sends $(\texttt{SET-DELAYS}, sid, \mathcal{T}')$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ at any round.
- If the adversary obtains a valid signature σ for the message $i||m||\eta$ by querying $\mathcal{F}_{\texttt{SIGN}}$ with $(\texttt{SIGN}, sid, i||m||\eta)$ then send $(\texttt{TIMED\_SIGN}, sid, +, m, \tau_{\texttt{req}})$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$. Upon receiving $(\texttt{TIMED\_SIGN}, sid, +, \mathsf{max}_{\mathsf{tsl}}, m, S, \tau_{\texttt{req}})$, check if $\mathsf{max}_{\mathsf{tsl}}+\delta \leq i$. If it is not, then ignore, otherwise set $\texttt{tsl}_{\texttt{post}} \leftarrow i$ and send $(\texttt{TIMED\_SIGNATURE}, sid, \texttt{t}, m, (\perp, \sigma, v, \texttt{tsl}_{\texttt{post}}))$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$. We refer to a signature $(m, (\perp, \sigma, \texttt{tsl}_{\texttt{post}}))$ computed in the way we just described as *predicted signature*. Any time that a predicted signature is caught by Sim, he stores it together with η.
- At any round Sim updates $\mathcal{T}$ according to the time table that is managed by $\mathcal{B}^{\mathsf{w}}$. That is, let $\mathcal{T}'$ be the time table of $\mathcal{B}^{\mathsf{w}}$, then then Sim sends $(\texttt{SET-DELAYS}, sid, \mathcal{T}')$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ at any round.
- Any time that $\mathcal{B}^{\mathsf{w}}$ issues a new value Sim checks if the signatures predicted by the adversary are actually valid. That is, for any recorded entry with the form $(m, (\perp, \sigma, \texttt{tsl}_{\texttt{post}}), \eta)$ that has $\texttt{tsl}_{\texttt{post}} = |\mathcal{H}|$ (we recall that $\mathcal{H}$ contains all the output issued by $\mathcal{B}^{\mathsf{w}}$), Sim checks if $\mathcal{H}[\texttt{tsl}_{\texttt{post}}] = \eta$. If it is not then Sim sends $I = (\texttt{DELETE}, sid, \texttt{t}, (\texttt{tsl}_{\texttt{back}}, x, \pi, \texttt{tsl}_{\texttt{post}}))$. In the end Sim aligns the size of the time-slots with the size new size of $\mathcal{H}$ by sending $(\texttt{NEW\_SLOT}, sid)$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$.

□

We assume that the parties $\mathcal{P}$ are registered to $\mathcal{G}_{\texttt{ledger}}$, $\mathcal{G}_{\texttt{clock}}$ and to the functionality $\mathcal{F}_{\sigma}^{\mathsf{w},+}$. Each party $p_i \in \mathcal{P}$ manages a local time table $\mathcal{T}_{p_i}^{\texttt{local}}$ that updates on any input he receives by invoking the procedure update_time_table().

Initialization. The signer $S \in \mathcal{P}$ sends $(\texttt{KEY\_GEN}, sid)$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ thus obtaining $(\texttt{VERIFICATION\_KEY}, sid, v)$.

Signature. The signer $S \in \mathcal{P}$ on input $(\texttt{TIMED\_SIGN}, \pm, sid, m, \tau_{\texttt{req}})$ executes the following steps.

1. Forward $(\texttt{TIMED\_SIGN}, sid, +, m, \tau_{\texttt{req}})$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$ and upon receiving the answer $(\texttt{TIMED\_SIGNATURE}, sid, +, (\perp, m, \sigma, \texttt{tsl}_{\texttt{post}}))$, create a transaction $\texttt{tx} \leftarrow (m, \sigma, \texttt{tsl}_{\texttt{post}})$ and send $(\texttt{SUBMIT}, sid, \texttt{tx})$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$.
2. Wait until $\texttt{tx}$ is added to the state of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Let $\texttt{tsl}_{\texttt{back}}$ be the block of state that contains $\texttt{tx}$, output $(\texttt{TIMED\_SIGNATURE}, sid, \pm, (\texttt{tsl}_{\texttt{back}}, m, \sigma, \texttt{tsl}_{\texttt{post}}))$.

Verification. The party $p_i \in \mathcal{P}$ on input $(\texttt{TIMED\_VERIFY}, sid, \pm, (\texttt{tsl}_{\texttt{back}}, m, \sigma, v', \texttt{tsl}_{\texttt{post}}))$ proceeds as follows.

1. Send $(\texttt{TIMED\_VERIFY}, sid, +, (\perp, m, \sigma, v', \texttt{tsl}_{\texttt{post}}))$ to $\mathcal{F}_{\sigma}^{\mathsf{w},+}$.
2. Upon receiving $(\texttt{TIMED\_VERIFIED}, sid, +, \perp, m, b, \tau)$ from $\mathcal{F}_{\sigma}^{\mathsf{w},+}$, if $b = 1$ then send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ else output $(\texttt{TIMED\_VERIFIED}, sid, \pm, \perp, m, 0, \perp)$.
3. Upon receiving the answer $(\texttt{READ}, sid, \mathsf{state})$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ check if the transaction $\texttt{tx} = (m, \sigma, \texttt{tsl}_{\texttt{post}})$ is stored in the $\texttt{tsl}_{\texttt{back}}$-th block of state. If it is, then find the smallest τ_{back} such that $\mathcal{T}_{p_i}^{\texttt{local}}[\tau_{back}] = \texttt{tsl}_{\texttt{back}}$ and output $(\texttt{TIMED\_VERIFIED}, sid, \tau_{back}, m, 1, \tau_{post})$ otherwise output $(\texttt{TIMED\_VERIFIED}, sid, \perp, m, 0, \perp)$.

Figure 5.5: The protocol $\Pi_{\sigma}^{\mathsf{w},\pm}$ in the $(\mathcal{F}_{\sigma}^{\mathsf{w},+}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model.

Theorem 10. *Let $\mathcal{F}_{\sigma}^{\mathsf{w},\pm}$ be the functionality parameterized by $\mathsf{w} = (\Delta, \texttt{MaxRound}, \texttt{WindowSize}, \texttt{waitingTime})$, then the protocol $\Pi_{\sigma}^{\mathsf{w},\pm}$ described in Fig. 5.5 realizes with perfect security $\mathcal{F}_{\sigma}^{\mathsf{w},\pm}$ in the $(\mathcal{F}_{\sigma}^{\mathsf{w},+}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model-hybrid model.*

5.5 Timed Zero Knowledge

TPoK with Postdate Security via UC-NIZK. Our protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ UC-realizes $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ in the $(\mathcal{F}_{\mathsf{NIZK}}, \mathcal{B}^{\mathsf{w}}, \mathcal{G}_{\texttt{clock}})$-hybrid model (with $\mathsf{w} = ((\mu, \ell), \texttt{MaxRound}, \texttt{WindowSize}, \texttt{MaxSize})$). $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ follows the *commit-and-prove* paradigm in which the prover commits to the witness w for the $\mathcal{NP}$-statement x to be proven, and then proves to the verifier that the committed message corresponds to a valid witness for x. In our protocol we want to associate some time-stamp to the proof generated by the prover, so we slightly modify the above approach as follows. The prover obtains (η, τ) by invoking the weak beacon $\mathcal{B}^{\mathsf{w}}$ and computes a commitment $\texttt{com}$ of $w||\eta$. Then the prover proves to the verifier that $\texttt{com}$ contains a witness for x concatenated with η. More precisely, let L be an $\mathcal{NP}$-language and $\mathcal{R}$ be the corresponding $\mathcal{NP}$-relation; to compute a proof the prover uses a NIZK PoK for the following $\mathcal{NP}$-relation

$$\mathsf{Rel}' = \{(\texttt{com}, x, \eta), (\mathsf{Dec}, w)\ s.t.\ \mathsf{Dec}(\texttt{com}, \mathsf{Dec}, \eta||w)) = 1 \text{ AND } (x, w) \in \mathcal{R}\}.$$

where x is the statement being proved and w is the corresponding witness (i.e. $(x, w) \in \mathcal{R}$). The verifier, upon receiving the proof computed by the prover accepts it if and only if the following two conditions hold: 1. value η has been output by $\mathcal{B}^{\mathsf{w}}$ in some round τ; 2. the NIZK proof given by the prover is accepting.

Since the NIZK that we use is a PoK and we assume that a malicious prover cannot predict the output of the weak beacon $\mathcal{B}^{\mathsf{w}}$ more than $\delta = \texttt{MaxRound} \cdot (\texttt{WindowSize} + \ell - \mu)$ rounds in advance then the verifier has the guarantee that the proof has been computed (and that the witness w was known by the prover) in some moment *subsequent* to $\tau - \delta$.

In Fig. 5.6 we show the details of our protocol. Our protocol uses $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{B}^{\mathsf{w}}$ and $\mathcal{G}_{\texttt{clock}}$. We recall that $\mathcal{B}^{\mathsf{w}}$ can be implemented by having read-only access to $\mathcal{W}_{\mathrm{WBU}}(\mathcal{G}_{\texttt{ledger}})$.

We assume that the parties $\mathcal{P}$ are registered to $\mathcal{F}_{\mathsf{NIZK}}$ for the $\mathcal{NP}$-relation Rel', $\mathcal{B}^{\mathsf{w}}$ and $\mathcal{G}_{\texttt{clock}}$.

Proof. The party $p \in \mathcal{P}$ on input $(\texttt{TIMED\_PROVE}, sid, (x, w), \tau)$ checks if $(x, w) \notin \mathcal{R}$. If it is, then ignores I else executes the following steps.

- Send $(\leftarrow\!\!\$, \tau, sid)$ to $\mathcal{B}^{\mathsf{w}}$ and upon receiving $(\leftarrow\!\!\$, sid, \texttt{tsl}, \eta)$, compute $(\texttt{com}, \mathsf{Dec}) \xleftarrow{\$} \mathsf{Com}(w||\eta)$ and define $x_t \leftarrow (\texttt{com}, x, \eta)$ and $w_t \leftarrow (w, \mathsf{Dec})$ such that $(x_t, w_t) \in \mathsf{Rel}'$.
- Send $(\texttt{PROVE}, sid, (x_t, w_t))$ to $\mathcal{F}_{\mathsf{NIZK}}$ and upon receiving the answer $(\texttt{PROOF}, sid, \pi)$ output $(\texttt{PROOF}, sid, x_t, (\pi, \texttt{tsl}))$.

Verification. The party p on input $(\texttt{TIMED\_VERIFY}, sid, +, x_t, (\perp, \pi, \texttt{tsl}))$ parses x_t as $(\texttt{com}, x, \eta)$ and executes the following steps.

- Query the beacon to see in which round τ the value η was issued.
- If this τ does not exist then output $(\texttt{TIMED\_VERIFICATION}, sid, +, \perp, 0, \perp)$, otherwise send $(\mathsf{Vf}, sid, x_t, \pi)$ to $\mathcal{F}_{\mathsf{NIZK}}$.
- Upon receiving $(\texttt{VERIFICATION}, sid, b)$ from $\mathcal{F}_{\mathsf{NIZK}}$ output $(\texttt{TIMED\_VERIFICATION}, sid, \texttt{t}, \perp, b, \tau)$.

Figure 5.6: The protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},+}$ in the $(\mathcal{F}_{\mathsf{NIZK}}, \mathcal{B}^{\mathsf{w}}, \mathcal{G}_{\texttt{clock}})$-hybrid model.

Theorem 11. *Let $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ be the functionality parameterized by $\mathsf{w} = (\Delta, \texttt{MaxRound}, \texttt{WindowSize}, \perp)$ then the protocol described in Fig. 5.6 realizes with perfect security $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ in the $(\mathcal{B}^{\mathsf{w}'}, \mathcal{F}_{\mathsf{NIZK}}, \mathcal{G}_{\texttt{clock}})$-hybrid model where $\mathcal{B}^{\mathsf{w}'}$ is parameterized by $\mathsf{w}' = ((\mu, \ell), \texttt{MaxRound}, \texttt{WindowSize}, \texttt{MaxSize})$ with $\Delta = \ell - \mu$.*

From Postdate Secure TPoK to Timed TPoK. In this section we show how to obtain a protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ that UC-realizes $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ in the $(\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model. $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ is the first protocol described in this work that uses $\mathcal{G}_{\texttt{ledger}}$ to store some information. Informally, every time that it is required to generate a proof for an $\mathcal{NP}$-statement x, $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ queries $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ thus obtaining a proof $(x, \pi, \texttt{tsl}_{\texttt{post}})$.

We assume that the parties $\mathcal{P}$ are registered to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ and to the functionality $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ that is parameterized with the $\mathcal{NP}$-relation $\mathcal{R}$. Every time that a party $p_i \in \mathcal{P}$ receives ad input she invokes the procedure update_time_table() (see Fig. 5.2 fore more details on this procedure).

Proof. The party $p_i \in \mathcal{P}$ on input $(\texttt{TIMED\_PROVE}, sid, \pm, (x, w), \tau)$ check if $(x, w) \in \mathcal{R}$. If it is not, then ignore I, proceed as follows otherwise.

1. Forward $(\texttt{TIMED\_PROVE}, sid, +, (x, w), \tau)$ to $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ and upon receiving the answer $(\texttt{TIMED\_PROOF}, sid, +, \perp, \pi, \texttt{tsl})$, create a transaction $\texttt{tx} \leftarrow (x, \pi, \texttt{tsl})$ and send $(\texttt{SUBMIT}, sid, \texttt{tx})$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$.
2. Wait until $\texttt{tx}$ is added to the state of $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$. Let $\texttt{tsl}_{\texttt{back}}$ be the block of state that contains $\texttt{tx}$, output $(\texttt{TIMED\_PROOF}, sid, \pm, x, (\texttt{tsl}_{\texttt{back}}, \pi, \texttt{tsl}_{\texttt{post}}))$.

Verification. The party $p_i \in \mathcal{P}$ on input $(\texttt{TIMED\_VERIFY}, sid, \pm, x, (\texttt{tsl}_{\texttt{back}}, \pi, \texttt{tsl}_{\texttt{post}}))$ proceeds as follows.

1. Send $(\texttt{TIMED\_VERIFY}, sid, +, x, (\perp, \pi, \texttt{tsl}_{\texttt{post}}))$ to $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$.
2. Upon receiving $(\texttt{TIMED\_VERIFICATION}, sid, +, \perp, b, \tau)$ from $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$, if $b = 1$ then send $(\texttt{READ}, sid)$ to $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ else output $(\texttt{TIMED\_VERIFICATION}, sid, \pm, \perp, 0, \perp)$.
3. Upon receiving the answer $(\texttt{READ}, sid, \mathsf{state})$ from $\mathcal{W}_{\text{WBU}}(\mathcal{G}_{\texttt{ledger}})$ check if the transaction $\texttt{tx} := (x, \pi, \texttt{tsl}_{\texttt{post}})$ is stored in the $\texttt{tsl}_{\texttt{back}}$-th block of state. If it is, then find the smallest τ_{back} such that $\mathcal{T}_{p_i}^{\texttt{local}}[\tau_{back}] = \texttt{tsl}_{\texttt{back}}$ and output $(\texttt{TIMED\_VERIFICATION}, sid, \pm, \tau_{back}, 1, \tau_{post}, 1)$ otherwise output $(\texttt{TIMED\_VERIFICATION}, sid, \pm, \perp, 0, \perp)$.

Figure 5.7: The protocol $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ in the $(\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}, \mathcal{G}_{\texttt{ledger}}, \mathcal{G}_{\texttt{clock}})$-hybrid model.

Then the prover of $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ generates a transaction that contains $(x, \pi, \texttt{tsl}_{\texttt{post}})$ and asks $\mathcal{G}_{\texttt{ledger}}$ to add the transaction to state. The parameter $\texttt{waitingTime}$ of $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ is the same parameter that in $\mathcal{G}_{\texttt{ledger}}$ defines the upper bound on the number of blocks that needs to be added to state before that a transaction submitted from an honest party gets into state. Intuitively, the *postdate security* of $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ comes immediately from the postdate security of $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},+}$ and the *backward security* comes from the fact that once that a transaction is part of state it cannot be removed. The formal construction of $\Pi_{\mathsf{TPoK}}^{\mathsf{w},\pm}$ for the $\mathcal{NP}$-Relation $\mathcal{R}$ is shown in Fig. 5.7. In this, every honest party p_i manages a table $\mathcal{T}_{p_i}^{\texttt{local}}$ that is updated on any input received by p_i according to the procedure update_time_table(), see Sec. 5.3.1 for more details. The proofs that our constructions implement the different notions of TPoK are very similar to the proof of Sec. 5.4.

5.5.1 Signature of Knowledge

It should be easy to see that postdate, backdate and timed secure NIZK implies respectively postdate, backdate and timed secure TSoK. Indeed we observe in the $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathsf{t}}$-hybrid model it is possible to obtain a protocol that UC-realizes $\mathcal{F}_{\mathsf{TSoK}}^{\mathsf{w},\mathsf{t}}$ following the approach proposed in [CL06] that we mentioned above. The only difference is that instead of using NIZK we rely on $\mathcal{F}_{\mathsf{TPoK}}^{\mathsf{w},\mathsf{t}}$ to prove that the ciphertext c contains the concatenation of the witness for x and the message m.

Chapter 6

Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings

Zero-knowledge proof systems have long been recognized as an important cryptographic primitive for protecting electronic privacy. The availability of a distributed ledger creates new possibilities for realizing strong privacy protection online. Private cryptographic currencies such as *zcash* are already deployed systems and other systems such as *zkay* extend smart contracts with privacy features. These developments are enabled by *s*uccinct *n*on-interactive *a*rguments of *k*nowledge (short SNARKs) which sacrifice perfect soundness for the benefit of a short non-interactive proof needed in the blockchain setting.

In this chapter we describe Sonic, a novel SNARK that is particularly well suited for distributed ledgers. The full version of our result can be found here [MBKM].

6.1 Introduction

In the decades since their introduction, zero-knowledge proofs have been used to support a wide variety of potential applications, ranging from verifiable outsourced computation [PHGR13, BCG+13, BCTV14, BCG+18] to anonymous credentials [CG08, BCC+09, GGM14, CKLM14, CDHK15b], with a multitude of other settings that also require a balance between privacy and integrity [BFG13, CCFG16, CKLM13, BBBF18, FPS+18]. In recent years, cryptocurrencies have been one increasingly popular real-world application [BCG+14, KMS+16, GM17, MS18], with general zero-knowledge protocols now deployed in both Zcash and Ethereum. In the cryptocurrency setting it is common for clients to download and verify every transaction published to the network. This means that small proof sizes and fast verification time are important for the practical deployment of zero-knowledge protocols. There are several practical schemes from which to choose, with a vast space of tradeoffs in performance and cryptographic assumptions.

Currently, the most attractive proving system from the verifier's perspective is a (pre-processing) succinct non-interactive argument of knowledge, or zk-SNARK for short, which has a small constant proof size and constant-time verification costs even for arbitrarily large relations. The most efficient scheme described in the literature is a zk-SNARK by Groth [Gro16] which contains only three group elements. Typically, zk-SNARKs require a trusted setup, a pairing-friendly elliptic curve, and rely on strong assumptions.

In contrast, proving systems such as Bulletproofs [BBB+18] do not require a trusted setup and depend on weaker assumptions. Unfortunately, although its proof sizes scale logarithmically with the relation size, Bulletproof verification time scales linearly, even when applying batching techniques. As a result, Bulletproofs are ideal for simpler relations.

Although zk-SNARKs have been deployed in applications, such as the private payment protocol in Zcash,

the trusted setup has emerged as a barrier for deployment. If the setup is compromised in Zcash, for example, an attacker could create counterfeit money without detection. It is possible to reduce risk by performing the setup with a multi-party computation (MPC) protocol, with the property that only one participant must be honest for the final parameters to be secure [Wil16, BCC+14]. However, the resulting parameters are specific to the individual relation, and so each distinct application must perform its own setup. Applications must also perform a new setup each time their construction changes, even for minor optimizations or bug fixes.

Groth et al. [GKM+18b] recently proposed a zk-SNARK scheme with a *universal* structured reference string (SRS[1]) that allows a single setup to support all circuits of some bounded size. Moreover, the SRS is *updatable*, meaning an open and dynamic set of participants can contribute secret randomness to it indefinitely. Although this is still a trusted setup in some sense, it increases confidence in the security of the parameters as only one previous contributor must have destroyed their secret randomness in order for the SRS to be secure.

In terms of efficiency, however, while the construction due to Groth et al. does have constant-size proofs and constant-time verification, it requires an SRS that is quadratic with respect to the number of multiplication gates in the supported arithmetic circuits. Moreover, updating the SRS requires a quadratic number of group exponentiations, and verifying the updates requires a linear number of pairings. Finally, while the prover and verifier need only a linear-size, circuit-specific string for a given fixed relation (rather than the whole SRS), deriving this from the SRS requires an expensive Gaussian elimination process. In a concrete setting such as Zcash, which has a circuit with 2^{17} multiplication gates, the SRS would be on the order of terabytes and is thus prohibitively expensive.

6.1.1 Results

We present Sonic, a new zk-SNARK for general arithmetic circuit satisfiability. Sonic requires a trusted setup, but unlike conventional SNARKs the structured reference string supports all circuits (up to a given size bound) and is also updatable, so that it can be continually strengthened. This addresses many of the practical challenges and risks surrounding such setups. Sonic's structured reference string is linear in size with respect to the size of supported circuits, as opposed to the scheme by Groth et al., which scales quadratically. The structured reference string in Sonic also does not need to be specialized or pre-processed for a given circuit. This makes a large, distributed and never-ending setup process a practical reality.

Proof verification in Sonic consists of a constant number of pairing checks. Unlike other zk-SNARKs, all proof elements are in the same source group, which has several advantages. Most significantly, when verifying many proofs at the same time, the pairing operations need to be computed only once. Thus the marginal costs stem solely from a handful of exponentiations in the group. We also remove the requirement for operations in the second source group, which are typically more expensive.

Sonic's verification includes checking the evaluation of a sparse bivariate polynomial in the scalar field. We introduce a method to check this evaluation succinctly (given a circuit-dependent precomputation) and thus maintain our zk-SNARK properties. Our proof of correct evaluation introduces a new permutation argument and a *grand-product* argument.

Additionally Sonic can achieve better concrete efficiency if an untrusted "helper" party aggregates a batch of proofs. This batching operation computes advice to speed up the verifier. In a blockchain application, this helper could be a miner-type client that already processes and verifies transactions for inclusion in the next block.

We define security in this setting in Section 6.2, and present and prove secure the regular usage of Sonic in Section 6.5 and Section 6.6. In Section 6.7 we present the more efficient version of Sonic which is helper-assisted. Finally, we implement our protocol and discuss its performance in Section 6.8, demonstrating verification times that are competitive with state-of-the-art pre-processing zk-SNARKs for typical arithmetic circuits. For any size of circuit proof sizes are 256 bytes and the verification times for circuits with small instances and arbitrarily sized witnesses are approximately 0.7ms (assuming there are helpers).

[1] "Structured reference string" is the recommended language to use when referring to what was once called a "common reference string" [Wor18].

6.1.2 Techniques

The goal of Sonic is to provide zero-knowledge arguments for the satisfiability of constraint systems representing NP-hard languages. Sonic defines its constraint system with respect to the two-variate polynomial equation used in Bulletproofs that was designed by Bootle et al. [BCC$^+$16]. In the Bulletproofs polynomial equation, there is one polynomial that is determined by the instance of the language and a second that is determined by the constraints. The polynomial determined by the instance $\boldsymbol{a}$ is given by

$$\sum_{i,j} a_{i,j} X^i Y^j$$

i.e., each element of the instance is used to scale a monomial in the overall polynomial. For this reason, an SRS that contains only hidden monomial evaluations suffices for committing to the instance. Groth et al. [GKM$^+$18b] showed that an SRS that contains monomials is updatable. The second polynomial that is determined by the constraints is known to the verifier. We use this knowledge to allow the verifier to obtain evaluations of the polynomial while avoiding putting constraint-specific secrets in the SRS.

To commit to our polynomials, we use a variation of a polynomial commitment scheme by Kate et al. [KZG10]. We prove the commitment scheme secure in the algebraic group model [FKL18], which is a model that lies somewhere between the standard model and the generic group model. This security proof does not follow from the initial reductions by Kate et al. because we additionally need to show that the adversary can extract the committed polynomials. Kate et al.'s scheme has constant size and verification time, but is designed for single-variate polynomials, whereas our polynomials are two-variate. To account for this, we hide only one evaluation point in the reference string. The polynomial defining the instance is of a special form where it can be committed to using a univariate scheme; i.e., it is of the form

$$\sum_{i} a_i X^i Y^i.$$

The prover first commits to the polynomial defining the statement, and then the second evaluation point y is determined in the clear. The prover can then commit to other polynomials of the form

$$\sum_{i,j} t_{i,j} X^i y^j$$

using a univariate scheme.

When the prover and verifier both know a two-variate polynomial that the verifier wants to calculate, this work can be unloaded onto the prover. In our scheme we utilize this observation by placing the work of computing the polynomial specifying the constraints onto the prover. The prover then has to show that the polynomial has been calculated correctly. We provide two methods of achieving this. In the first, we simply provide a proof that the evaluation is correct. While asymptotically preferable, concretely this proof is three times the size of our second method. In this scenario, many proofs are calculated by many provers, and then a "helper" calculates the circuit-specifying polynomial for each proof. The circuit-specifying polynomial contains no private information, so the helper can be run by anyone. The helper then proves that they have calculated all of the polynomials correctly at the same time, which they can do succinctly with a one-off circuit-dependent cost that can be amortized over many proofs.

6.2 Definitions for Updatable Reference Strings

In this section, we revisit the definitions around updatable SRS schemes due to Groth et al. [GKM$^+$18b], in terms of defining properties of zero-knowledge proofs in the case in which the adversary may subvert or participate in the generation of the common reference string. Given that our protocol in Section 6.5 is interactive (but made non-interactive in the random oracle model), we also present new definitions for interactive protocols that take into account these alternative methods of SRS generation.

6.2.1 Notation

If x is a binary string then $|x|$ denotes its bit length. If S is a finite set then $|S|$ denotes its size and $x \xleftarrow{\$} S$ denotes sampling a member uniformly from S and assigning it to x. We use $\lambda \in \mathbb{N}$ to denote the security parameter and 1^λ to denote its unary representation. We use ε to denote the empty string.

Algorithms are randomized unless explicitly noted otherwise. "PPT" stands for "probabilistic polynomial time" and "DPT" stands for "deterministic polynomial time." We use $y \leftarrow A(x; r)$ to denote running algorithm A on inputs x and random coins r and assigning its output to y. We write $y \xleftarrow{\$} A(x)$ or $y \xleftarrow{r} A(x)$ (when we want to refer to r later on) to denote $y \leftarrow A(x; r)$ for r sampled uniformly at random.

We use code-based games in security definitions and proofs [BR06]. A game $\mathsf{Sec}_{\mathcal{A}}(\lambda)$, played with respect to a security notion Sec and adversary $\mathcal{A}$, has a MAIN procedure whose output is the output of the game. The notation $\mathsf{Pr}[\mathsf{Sec}_{\mathcal{A}}(\lambda)]$ is used to denote the probability that this output is 1.

6.2.2 The Subvertible SRS Model

Intuitively, the subvertible SRS model [BFS16] allows the adversary to fully generate the reference string itself, and the updatable SRS model [GKM+18b] allows the adversary to partially contribute to its generation by performing some update. Formally, an updatable SRS scheme is defined by two PPT algorithms Setup and Update, and a DPT algorithm $\mathsf{VerifySRS}$. These behave as follows:

- $(\mathtt{srs}, \rho) \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$ takes as input the security parameter and returns a SRS and proof of its correctness.
- $(\mathtt{srs}', \rho') \xleftarrow{\$} \mathsf{Update}(1^\lambda, \mathtt{srs}, (\rho_i)_{i=1}^n)$ takes as input the security parameter, a SRS, and a list of update proofs. It outputs an updated SRS and a proof of the correctness of the update.
- $b \leftarrow \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, (\rho_i)_{i=1}^n)$ takes as input the security parameter, a SRS, and a list of proofs. It outputs a bit indicating acceptance ($b = 1$), or rejection ($b = 0$).

We consider an updatable SRS to be perfectly correct if an honest updater always convinces an honest verifier.

Definition 16. An updatable SRS scheme is perfectly correct if

$$\Pr\left[(\mathtt{srs}, \rho) \xleftarrow{\$} \mathsf{Setup}(1^\lambda) : \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, \rho) = 1\right] = 1,$$

and if for all $(\lambda, \mathtt{srs}, (\rho_i)_{i=1}^n)$ where $\mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, (\rho)_{i=1}^n) = 1$, we have that

$$\Pr\left[\begin{array}{l}(\mathtt{srs}', \rho_{n+1}) \xleftarrow{\$} \mathsf{Update}(1^\lambda, \mathtt{srs}, (\rho_i)_{i=1}^n) : \\ \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}', (\rho)_{i=1}^{n+1}) = 1\end{array}\right] = 1.$$

In terms of the usage of these SRSs, a protocol cannot satisfy both subvertible zero-knowledge and subvertible soundness [BFS16]. That is, assuming the adversary knows all the randomness used to generate the SRS, they can either break the zero-knowledge property of the scheme or they can break the soundness property of the scheme. We thus recall here the two strongest properties we can hope to satisfy, which are subvertible zero-knowledge and updatable knowledge soundness. The definitions of these properties are simplified versions of the ones given by Groth et al. [GKM+18b], with the addition of a random oracle H (which behaves as expected, so we omit its description).

Let R be a polynomial-time decidable relation with triples $(\mathtt{srs}, \phi, w)$. We say w is a witness to the instance ϕ being in the relation defined by $\mathtt{srs}$ when $(\mathtt{srs}, \phi, w) \in R$. We consider an argument $(\mathsf{Prove}, \mathsf{Verify})$ to be subversion zero-knowledge if an adversarial verifier, including one that (fully) generates the SRS, cannot differentiate between real and simulated proofs.

Definition 17 (Subvertible Zero-Knowledge)**.** An argument for the relation R is S-*zero-knowledge* if for all PPT algorithms $\mathcal{A}$ there exists a PPT extractor $\mathcal{X}$ and a simulator $\mathsf{SimProve}$ such that the advantage $|2\Pr[\mathsf{S\text{-}ZK}_{\mathcal{A},\mathcal{X}}(1^\lambda)] - 1|$ is negligible in λ, where this game is defined as follows:

$\underline{\text{MAIN } \mathsf{S\text{-}ZK}_{\mathcal{A},\mathcal{X}_\mathcal{A}}(\lambda)}$
$b \xleftarrow{\$} \{0,1\}$
$(\mathtt{srs}, (\rho_i)_{i=1}^n) \xleftarrow{r} \mathcal{A}^H(1^\lambda)$
$\tau \xleftarrow{\$} \mathcal{X}_\mathcal{A}(r)$
if $\mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, (\rho_i)_{i=1}^n) = 0$ return 0
$b' \leftarrow \mathcal{A}^{H,\mathcal{O}_{\mathsf{pf}}}(r)$
return $b' = b$

$\underline{\mathcal{O}_{\mathsf{pf}}(\phi, w)}$
if $(\mathtt{srs}, \phi, w) \notin R$ return $\bot$
if $b = 0$ return $\mathsf{SimProve}(\mathtt{srs}, \tau, \phi)$
else return $\mathsf{Prove}(\mathtt{srs}, \phi, w)$

To define update knowledge-soundness, we consider an adversary that can influence the generation of the SRS. To do this, it can query an oracle with an intent set to "setup" (for the first update proof), "update" (for all subsequent update proofs), or "final" (to signal the SRS for which it will attempt to forge proofs). The oracle sets the SRS only if: (1) all update proofs verify; and (2) it was responsible for generating at least one of the update proofs. We do not use updatable knowledge soundness directly, but this part of the security game (in which $\mathcal{A}$ and $\mathsf{U}\text{-}\mathcal{O}_\mathsf{s}$ interact to create the SRS) can be re-purposed for any cryptographic primitive. We use this updatability notion mainly for the polynomial commitment scheme we present in Section 6.5.2.

Definition 18 (Updatable Knowledge Soundness)**.** An argument for the relation R is U-*knowledge-sound* if for all PPT algorithms $\mathcal{A}$ there exists a PPT extractor $\mathcal{X}_\mathcal{A}$ such that $\Pr[\mathsf{U\text{-}KSND}_{\mathcal{A},\mathcal{X}_\mathcal{A}}(1^\lambda)]$ is negligible in λ, where this game is defined as follows:

$\underline{\text{MAIN } \mathsf{U\text{-}KSND}_{\mathcal{A},\mathcal{X}_\mathcal{A}}(\lambda)}$
$\mathtt{srs} \leftarrow \bot$
$(\phi, \pi) \xleftarrow{r} \mathcal{A}^{H,\mathsf{U}\text{-}\mathcal{O}_\mathsf{s}}(1^\lambda)$
$w \xleftarrow{\$} \mathcal{X}_\mathcal{A}(\mathtt{srs}, r)$
return $\mathsf{Verify}(\mathtt{srs}, \phi, \pi) \ \wedge \ (\mathtt{srs}, \phi, w) \notin R$

$\underline{\mathsf{U}\text{-}\mathcal{O}_\mathsf{s}(\mathsf{intent}, \mathtt{srs}_n, (\rho_i)_{i=1}^n)}$
if $\mathtt{srs} \neq \bot$ return $\bot$
if $\mathsf{intent} = \mathsf{setup}$
  $(\mathtt{srs}', \rho') \xleftarrow{\$} \mathsf{Setup}(1^\lambda)$
  $Q \leftarrow Q \cup \{\rho'\}$
  return $(\mathtt{srs}', \rho')$
if $\mathsf{intent} = \mathsf{update}$
  $b \leftarrow \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}_n, (\rho_i)_{i=1}^n)$
  if $b = 0$ return $\bot$
  $(\mathtt{srs}', \rho') \xleftarrow{\$} \mathsf{Update}(1^\lambda, \mathtt{srs}_n, (\rho_i)_{i=1}^n)$
  $Q \leftarrow Q \cup \{\rho'\}$
  return $(\mathtt{srs}', \rho')$
if $\mathsf{intent} = \mathsf{final}$
  $b \leftarrow \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}_n, (\rho_i)_{i=1}^n)$
  if $b = 0$ or $Q \cap \{\rho_i\}_i = \emptyset$ return $\bot$
  $\mathtt{srs} \leftarrow \mathtt{srs}_n$; return $\mathtt{srs}$
else return $\bot$

To argue about the soundness of Sonic, we consider an interactive definition. We do not use the standard definition of special soundness because our verifier provides two challenges, but rather the generalized notion of *witness-extended emulation* [Lin03]. We adapt the definition given by Bootle et al. [BCC+16] as follows:

Definition 19. Let P be an argument for the relation R. Then it satisfies *updatable witness-extended emulation*

if for all DPT P^* there exists an expected polynomial-time (PT) emulator $\mathcal{E}$ such that for all PPT algorithms $\mathcal{A}$:

$$\begin{aligned}
&\mathsf{Pr}[(\mathtt{srs}', \rho') \xleftarrow{\$} \mathsf{Setup}(1^\lambda) ; \\
&\quad (\mathtt{srs}, (\rho_i)_i, \phi, w) \xleftarrow{\$} \mathcal{A}(\mathtt{srs}', \rho') ; \\
&\quad \mathsf{view} \leftarrow \langle \mathsf{P}^*(\mathtt{srs}, \phi, w), \mathsf{V}(\mathtt{srs}, \phi) \rangle \ : \\
&\quad \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, (\rho_i)_i) \wedge \mathcal{A}(\mathsf{view}) = 1] \\
\approx\ &\mathsf{Pr}[(\mathtt{srs}', (\rho'_i)_i) \xleftarrow{\$} \mathsf{Setup}(1^\lambda) ; \\
&\quad (\mathtt{srs}, (\rho_i)_{i=1}^n, \phi, w) \xleftarrow{\$} \mathcal{A}(\mathtt{srs}', \rho') ; \\
&\quad (\mathsf{view}, w) \leftarrow \mathcal{E}^{\langle \mathsf{P}^*(\mathtt{srs}, \phi, w), \mathsf{V}(\mathtt{srs}, \phi) \rangle} \ : \\
&\quad \mathsf{VerifySRS}(1^\lambda, \mathtt{srs}, (\rho_i)_i) \wedge \mathcal{A}(\mathsf{view}) = 1 \wedge \\
&\quad \text{if view is accepting then } (\phi, w) \in R],
\end{aligned}$$

where the oracle called by $\mathcal{E}^{\langle \mathsf{P}^*(\mathtt{srs}, \phi, w), \mathsf{V}(\mathtt{srs}, \phi) \rangle}$ permits rewinding to a specific point and resuming with fresh randomness for the verifier from this point onwards.

This definition uses a slightly different setup from the one in Definition 18: rather than interacting arbitrarily with an update oracle to set the SRS, the adversary is instead given an initial one and is then allowed to update that in a one-shot fashion. Following Groth et al. [GKM+18b, Lemma 6], these two definitions are equivalent for Sonic, so we opt for the simpler one.

6.3 Building Blocks

6.3.1 Bilinear Groups

Let $\mathsf{BilinearGen}(1^\lambda)$ be a bilinear group generator that given the security parameter 1^λ produces bilinear parameters $bp = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h)$, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are groups of prime order p with generators $g \in \mathbb{G}_1$, $h \in \mathbb{G}_2$ and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerative bilinear map. That is, $e(g^a, h^b) = e(g, h)^{ab}\ \forall a, b \in \mathbb{F}_p$ and $e(g, h)$ generates $\mathbb{G}_T$.

We require bilinear groups such that the maximum size of our circuit is bounded by $d^2 \leq (p-1)/32$. In practice we expect that $d^2 \ll (p-1)/32$.

We employ bilinear group generators that produce what Galbraith, Paterson and Smart [GPS08] classify as Type III bilinear groups. For such groups no efficiently computable homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$ exist. These are currently the most efficient bilinear groups.

6.3.2 The Algebraic Group Model

Sonic is proven secure in the algebraic group model (AGM) by Fuchsbauer et al [FKL18], who used it to prove (among other things) that Groth's 2016 scheme [Gro16] is secure under a "q-type" variant of the discrete log assumption. Previously the only security proof for this scheme was provided in the generic group model (GGM). Although proofs in the GGM can increase our confidence in the security of a scheme, its scope is limited since it does not capture group-specific algorithms that make use of its representation (such as index calculus approaches).

The AGM lies between the standard model and the GGM, and it is a restricted model of computation that covers group-specific attacks while allowing a meaningful security analysis. Adversaries are assumed to be

restricted in the sense that they can output only group elements obtained by applying the group operation to previously received group elements. Unlike the GGM, in the AGM one proves security implications via reductions to assumptions (just as in proofs in the standard model).

It is so far unknown how the AGM relates to knowledge-of-exponent (KOE) assumptions, which have been used to build every known SNARK that has been proven secure in the standard model (and indeed it is known that SNARKs cannot be proven secure under more standard falsifiable assumptions [GW11]). The format of these KOE assumptions is similar to the AGM in the sense that proving the assumption incorrect would require showing that there is an adversary that can compute group elements of a given format but that cannot extract an algebraic representation. Popular KOE assumptions in asymmetric bilinear groups all require the adversary to compute elements in the second source group. As we would like to avoid introducing proof elements in the second source group (as these are typically more expensive due to current implementations of asymmetric bilinear groups), we instead decided to work with the AGM.

An algorithm $\mathcal{A}_{\mathsf{alg}}$ is called algebraic if whenever it outputs an element Z in $\mathbb{G}$, it also outputs a representation $(z_1, \ldots, z_t) \in \mathbb{F}_p^t$ such that $Z = \prod_{i=1}^{t} g_i^{z_i}$ where $\mathcal{L} = \{g_1, \ldots, g_t\}$ is the list of all group elements given to $\mathcal{A}_{\mathsf{alg}}$ in its execution thus far. Unlike the GGM, in the AGM one proves security implications via reductions. To prove our scheme secure in the algebraic group model we use the q discrete log assumption (q-DLOG), as follows:

Assumption 2 (q-DLOG assumption). Suppose that $\mathcal{A}$ is an algebraic adversary. Then

$$\Pr\left[\begin{array}{l} bp \leftarrow \mathsf{BilinearGen}(1^\lambda);\ x \xleftarrow{\$} \mathbb{F}_p; \\ x' \xleftarrow{\$} \mathcal{A}(bp, \{g^{x^i}, h^{x^i}\}_{i=-q}^{q}) \ :\ x = x' \end{array}\right]$$

is negligible in 1^λ.

6.3.3 Structured Reference String

In all of the following we require a structured reference string with unknowns x and α of the following form

$$\left\{ \{g^{x^i}\}_{i=-d}^{d},\ \{g^{\alpha x^i}\}_{i=-d, i\neq 0}^{d},\ \{h^{x^i}, h^{\alpha x^i}\}_{i=-d}^{d},\ e(g,h^\alpha) \right\}$$

for some large enough d to support the circuit depth n.

This string is designed so that g^α is omitted from the reference string. Thus we can, when necessary, force the prover to demonstrate that a committed polynomial (in x) has a zero constant term.

6.3.4 Polynomial Commitment Scheme

Sonic uses two main primitives as building blocks: a polynomial commitment scheme and a signature of correct computation. A polynomial commitment scheme is defined by three DPT protocols:

- $F \leftarrow \mathsf{Commit}(bp, \mathtt{srs}, \mathsf{max}, f(X))$ takes as input the bilinear group, the structured reference string, a maximum degree, and a Laurent polynomial with powers between $-d$ and max. It returns a commitment F.
- $(f(z), W) \leftarrow \mathsf{Open}(bp, \mathtt{srs}, \mathsf{max}, F, z, f(X))$ takes as input the same parameters as the commit algorithm in addition to a commitment F and a point in the field z. It returns an evaluation $f(z)$ and a proof of its correctness.
- $b \leftarrow \mathsf{pcV}(bp, \mathtt{srs}, \mathsf{max}, F, z, v, W)$ takes as input the bilinear group, the SRS, a maximum degree, a commitment, a point in the field, an evaluation and a proof. It outputs a bit indicating acceptance ($b = 1$), or rejection ($b = 0$).

We require that this scheme is *evaluation binding*; i.e., given a commitment F, an adversary cannot open F to two different evaluations v_1 and v_2 (more formally, that it cannot output a tuple $(F, z, v_1, v_2, W_1, W_2)$ such that pcV returns 1 on both sets of evaluations and proofs). We also require that it is *bounded polynomial extractable*; i.e., any adversary that can provide a valid evaluation opening also knows an opening $f(X)$ with powers $-d \leq i \leq \mathsf{max}, i \neq d - \max$ (more formally, that this is true for any adversary that outputs a tuple (F, z, v, W) that passes verification). For both properties, we require that they hold with respect to an adversary that can update the SRS; i.e., that has access initially to the oracle in Definition 18.

In Section 6.5.2 we provide a polynomial commitment scheme satisfying these two properties. We prove its security in the algebraic group model in Theorem 14.

6.3.5 Signature of Correct Computation

A signature of correct computation is defined by two DPT protocols:

- $(s(z, y), \mathsf{sc}) \leftarrow \mathsf{scP}(bp, \mathtt{srs}, s(X, Y), (z, y))$ takes as input the bilinear group, the SRS, a two-variate polynomial $s(X, Y)$, and two points in the field (z, y). It returns an evaluation $s(z, y)$ and a proof sc.
- $b \leftarrow \mathsf{scV}(bp, \mathtt{srs}, s(X, Y), (z, y), s, \mathsf{sc})$ takes as input the same parameters as the scP algorithm in addition to an evaluation and a proof. It outputs a bit indicating acceptance ($b = 1$), or rejection ($b = 0$).

We require that this scheme is sound; i.e., given (z, y) and s, an adversary can convince the verifier only if $s = s(z, y)$.

We provide two competing constructions: one in Section 6.7 and the other in Section 6.6. The first has linear verifier computation, but can be aggregated by an untrusted helper to achieve constant verifier computation in the batched setting. The second has constant verifier computation but higher concrete overhead. Both constructions have constant size.

6.4 System of Constraints

Sonic represents circuits using a form of constraint system proposed by Bootle et al. [BCC$^+$16]. We make several modifications so that their approach is practical in our setting.

Our constraint system has three vectors of length n: **a**, **b**, **c** representing the left inputs, right inputs, and outputs of multiplication constraints respectively, so that

$$\mathbf{a} \circ \mathbf{b} = \mathbf{c}.$$

We also have Q linear constraints of the form

$$\mathbf{a} \cdot \mathbf{u_q} + \mathbf{b} \cdot \mathbf{v_q} + \mathbf{c} \cdot \mathbf{w_q} = k_q$$

where $\mathbf{u_q}, \mathbf{v_q}, \mathbf{w_q} \in \mathbb{F}^n$ are fixed vectors for the q-th linear constraint, with instance value $k_q \in \mathbb{F}_p$. For example, to represent the constraint $x^2 + y^2 = z$, one would set

- $\boldsymbol{a} = (x, y)$, $\boldsymbol{b} = (x, y)$, $\boldsymbol{c} = (x^2, y^2)$
- $\boldsymbol{u}_1 = (1, 0), \boldsymbol{v}_1 = (-1, 0), \boldsymbol{w}_1 = (0, 0), k_1 = 0$
- $\boldsymbol{u}_2 = (0, 1), \boldsymbol{v}_2 = (0, -1), \boldsymbol{w}_2 = (0, 0), k_2 = 0$
- $\boldsymbol{u}_3 = (0, 0), \boldsymbol{v}_3 = (0, 0), \boldsymbol{w}_3 = (1, 1), k_3 = z$

Any arithmetic circuit can be represented with our constraint system by using the multiplication constraints to determine the multiplication gates and the linear constraints to determine the wiring of the circuit and the addition gates. Thus the constraint system covers NP.

We proceed to compress the n multiplication constraints into an equation in formal indeterminate Y, as

$$\sum_{i=1}^{n}(a_i b_i - c_i)Y^i = 0.$$

In order to support our later argument, we (redundantly) encode these constraints into negative exponents of Y, as

$$\sum_{i=1}^{n}(a_i b_i - c_i)Y^{-i} = 0.$$

We compress the Q linear constraints similarly, scaling by Y^n to preserve linear independence.

$$\sum_{q=1}^{Q}\left(\mathbf{a}\cdot\mathbf{u_q} + \mathbf{b}\cdot\mathbf{v_q} + \mathbf{c}\cdot\mathbf{w_q} - k_q\right)Y^{q+n} = 0.$$

Let us define the polynomials

$$\begin{aligned}
u_i(Y) &= \sum_{q=1}^{Q} Y^{q+n}u_{q,i} \\
v_i(Y) &= \sum_{q=1}^{Q} Y^{q+n}v_{q,i} \\
w_i(Y) &= -Y^i - Y^{-i} + \sum_{q=1}^{Q} Y^{q+n}w_{q,i} \\
k(Y) &= \sum_{q=1}^{Q} Y^{q+n}k_q
\end{aligned}$$

and combine our multiplicative and linear constraints to form the equation

$$\mathbf{a}\cdot\mathbf{u}(Y) + \mathbf{b}\cdot\mathbf{v}(Y) + \mathbf{c}\cdot\mathbf{w}(Y) + \sum_{i=1}^{n} a_i b_i (Y^i + Y^{-i}) - k(Y) = 0. \tag{6.1}$$

Given a choice of $(\mathbf{a}, \mathbf{b}, \mathbf{c}, k(Y))$, we have that Equation 6.1 holds at all points if the constraint system is satisfied. If the constraint system is not satisfied the equation fails to hold with high probability, given a large enough field.

We apply a technique from Bootle et al. [BCC$^+$16] to embed the left hand side of Equation 6.1 into the constant term of a polynomial $t(X, Y)$ in a second formal indeterminate X. We design the polynomial $r(X, Y)$ such that $r(X, Y) = r(XY, 1)$.

$$\begin{aligned}
r(X,Y) &= \sum_{i=1}^{n}\left(a_i X^i Y^i + b_i X^{-i}Y^{-i} + c_i X^{-i-n}Y^{-i-n}\right) \\
s(X,Y) &= \sum_{i=1}^{n}\left(u_i(Y)X^{-i} + v_i(Y)X^i + w_i(Y)X^{i+n}\right) \\
r'(X,Y) &= r(X,Y) + s(X,Y) \\
t(X,Y) &= r(X,1)r'(X,Y) - k(Y)
\end{aligned}$$

The coefficient of X^0 in $t(X, Y)$ is the left-hand side of Equation 6.1. Sonic demonstrates that the constant term of $t(X, Y)$ is zero, thus demonstrating that our constraint system is satisfied.

6.5 The Basic Sonic Protocol

Sonic is a zero-knowledge argument of knowledge that allows a prover to demonstrate that a constraint system (described in Section 6.4) is satisfied for a hidden witness $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and for known instance $\mathbf{k}$. The instance $\mathbf{k}$ is uploaded into the constraint system through the polynomial $k(Y)$. Given a choice of $r(X, Y)$ from Section 6.4, if for random $y \in \mathbb{F}_p$ we have that the constant term of $t(X, y)$ is zero, the constraint system is satisfied with high probability.

Our Sonic protocol is built directly from a polynomial commitment scheme and a signature of correct computation, as visualized in Figure 6.1. We discuss here the basic Sonic protocol, assuming these building blocks are in place, and provide a suitable bounded extractable polynomial commitment scheme in Section 6.5.2 that we prove secure in the AGM. In Sections 6.6 and 6.7 we discuss two different methods of constructing the signature of correct computation, one which gives rise to a standalone zk-SNARK and one which achieves better practical results through the use of an untrusted helper.

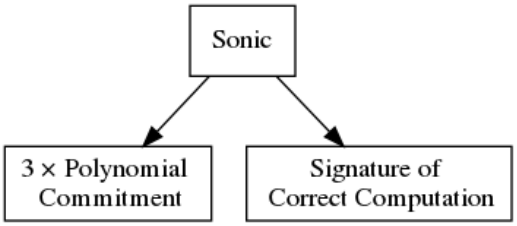

Figure 6.1: The basic Sonic protocol is built on top of a bounded-extractable polynomial commitment scheme and a signature of correct computation.

Our protocol begins by having the prover construct $r(X, Y)$ using their hidden witness. They commit to $r(X, 1)$, setting the maximum degree to n. The verifier sends a random challenge y. The prover commits to $t(X, y)$, and our commitment scheme ensures that this polynomial has no constant term. The verifier sends a second challenge z. The prover opens their committed polynomials to $r(z, 1)$, $r(z, y)$ and $t(z, y)$. The verifier can calculate $r'(z, y)$ for itself from these values and thus can check that $r(z, y)r'(z, y) - k(y) = t(z, y)$. Note that the coefficients of the public polynomial $k(Y)$ are determined by the instance that the prover is claiming is in the language. If this holds then the verifier learns that the evaluated polynomials were computed by a prover that knows a valid witness. A more formal description of this protocol is given in Figure 6.2.

The verifier's check that the quadratic polynomial equation is satisfied is performed in the field. This means we avoid having proof elements on both sides of the pairing, which is useful for efficiency, without contradicting Groth's result about Non-Interactive Linear Proofs (NILPs) requiring a quadratic constraint [Gro16]. As a result, when batching we avoid having to check one pairing equation per proof (pairing operations are expensive) and can instead check one field equation per proof.

The Fiat-Shamir transformation takes an interactive argument and replaces the verifier challenges with the output of a hash function. The idea is that the hash function will produce random-looking outputs and therefore be a suitable replacement for the verifier. We describe Sonic in the interactive setting where all verifier challenges are random field elements. In practice we assume that the Fiat-Shamir heuristic would be applied in order to obtain a non-interactive zero-knowledge argument in the random oracle model.

Theorem 12. *Assuming the ability to extract a trapdoor for the subverted reference string, Sonic satisfies subversion zero-knowledge.*

Proof. To prove subversion zero-knowledge, we need to both show the existence of an extractor $\mathcal{X}_{\mathcal{A}}$ that can compute a trapdoor, and describe a SimProve algorithm that produces indistinguishable proofs when provided with the extracted trapdoor. We do not discuss the details of SRS generation here so we do not prove the existence of the extractor, but one such example can be found in the original proof of Groth et al. [GKM$^+$18b, Lemma 4].

Common input: $\mathsf{info} = bp,\ \mathtt{srs},\ s(X,Y),\ k(Y),\ e(g,h^{\alpha})$
Prover's input: $\mathbf{a}, \mathbf{b}, \mathbf{c}$

$\underline{\mathsf{zkP}_1(\mathsf{info}, \mathbf{a}, \mathbf{b}, \mathbf{c}) \mapsto R}$:
$c_{n+1}, c_{n+2}, c_{n+3}, c_{n+4} \xleftarrow{\$} \mathbb{F}_p$
$r(X,Y) \leftarrow r(X,Y) + \sum_{i=1}^{4} c_{n+i} X^{-2n-i} Y^{-2n-i}$
$R \leftarrow \mathsf{Commit}(bp, \mathtt{srs}, n, r(X,1))$
send R

$\underline{\mathsf{zkV}_1(\mathsf{info}, R) \mapsto y}$:
send $y \xleftarrow{\$} \mathbb{F}_p$

$\underline{\mathsf{zkP}_2(y) \mapsto T}$:
$T \leftarrow \mathsf{Commit}(bp, \mathtt{srs}, d, t(X,y))$
send T

$\underline{\mathsf{zkV}_2(T) \mapsto z}$:
send $z \xleftarrow{\$} \mathbb{F}_p$

$\underline{\mathsf{zkP}_3(z) \mapsto (a, W_a, b, W_b, W_t, s, \mathsf{sc})}$:
$(a = r(z,1), W_a) \leftarrow \mathsf{Open}(R, z, r(X,1))$
$(b = r(z,y), W_b) \leftarrow \mathsf{Open}(R, yz, r(X,1))$
$(t = t(z,y), W_t) \leftarrow \mathsf{Open}(T, z, t(X,y)))$
$(s = s(z,y), \mathsf{sc}) \leftarrow \mathsf{scP}(\mathsf{info}, s(X,Y), (z,y))$
send $(a, W_a, b, W_b, W_t, s, \mathsf{sc})$

$\underline{\mathsf{zkV}_3(a, W_a, b, W_b, W_t, s, \mathsf{sc}) \mapsto 0/1}$:
$t \leftarrow a(b+s) - k(y)$
check $\mathsf{scV}(\mathsf{info}, s(X,Y), (z,y), (s, \mathsf{sc}))$
check $\mathsf{pcV}(bp, \mathtt{srs}, n, R, z, (a, W_a))$
check $\mathsf{pcV}(bp, \mathtt{srs}, n, R, yz, (b, W_b))$
check $\mathsf{pcV}(bp, \mathtt{srs}, d, T, z, (t, W_t))$
return 1 if all checks pass, else return 0

Figure 6.2: The interactive Sonic protocol to check that the prover knows a valid assignment of the wires in the circuit. The stated algorithms describe the individual steps of each of the parties (e.g., zkV_i describes the i-th step of the verifier given the output of zkP_{i-1}), and both parties are assumed to keep state for the duration of the interaction.

The simulator is given the trapdoor g^α and chooses random vectors $\boldsymbol{a}$, $\boldsymbol{b}$ from $\mathbb{F}_p$ of length n and sets $\boldsymbol{c} = \boldsymbol{a} \cdot \boldsymbol{b}$. It computes $r(X, Y)$, $r'(X, Y)$, $t(X, Y)$ as in Section 6.4 where (unlike for the prover) $t(X, Y)$ can have a non-zero coefficient in X^0. The simulator then behaves exactly as the prover in Figure 6.2 with its random polynomials.

Both the prover and the simulator evaluate $g^{r(x,1)}$, $r(z, 1)$, and $r(zy, 1)$. This reveals 3 evaluations (some of these are in the exponent). The prover has four blinders for $r(X)$ with respect to the powers $-2n - 1, -2n - 2, -2n - 3, -2n - 4$. Thus for a verifier that obtains less than three evaluations, the prover's polynomial is indistinguishable from the simulator's random polynomial. All other components in the proofs are either uniquely determined given the previous components for both prover and simulator, or are calculated independently from the witness (and are chosen in the same method by both prover and simulator). □

Theorem 13. *Sonic has witness extended emulation, when instantiated using a secure polynomial commitment scheme and a sound signature of correct computation.*

Proof. Soundness of the signature of correct computation gives us that $s = s(z, y)$.

Bounded polynomial extractability tells us that R contains the polynomial

$$r(X, 1) = \sum_{i=-d, i\neq -d+n}^{n} r_i X^i$$

and that T contains the polynomial

$$\tau(X) = \sum_{i=-d, i\neq 0}^{d} \tau_i X^i.$$

Observe that in our polynomial constraint system $3n < d$ (otherwise we cannot commit to $t(X, Y)$), thus $r(X, Y)$ has no $-d + n$ term.

We show that the element T can be computed only if the circuit is satisfied by the polynomial coefficients extracted from R. Evaluation binding tells us that $a = r(z, 1)$, $b = r(zy, 1) = r(z, y)$ and the verifier checks that $t = a(b + s) - k(y) = \tau(z)$. Suppose this holds for $n + Q + 1$ different challenges $y \in \mathbb{F}_p$. Then we have equality of polynomials in Section 6.4 since a non-zero polynomial of degree $n+Q+1$ cannot have $n+Q$ roots; i.e.,

$$r(X)(r(X, Y) + s(X, Y)) - k(Y)$$

has no constant term. This implies that $r(X, y)$ defines a valid witness. □

6.5.1 Efficiency

As seen in Figure 6.2, our prover uses two polynomial commitments which it opens at three points. It also uses one signature of correct computation. Two of these openings can be batched using techniques we describe in the full version [MBKM]. The idea behind the batching is that given two polynomial commitments F_1 and F_2, if a verifier chooses random values r_1 and r_2, then an adversary can open $F_1^{r_1} F_2^{r_2}$ only if it can also (with high probability) open F_1 and F_2 separately. The polynomial $k(Y)$ is sparse and determined by the instance, and thus takes $\mathcal{O}(\ell)$ field operations to compute.

6.5.2 Polynomial Commitment Scheme

Sonic uses a polynomial commitment scheme which is an adaptation of a scheme by Kate, Zaverucha, and Goldberg [KZG10]. This scheme has constant-sized proofs for any size polynomial and verification consists of checking a single pairing. We require that the scheme is evaluation binding; i.e., given a commitment F, an adversary cannot open F to two different evaluations v_1 and v_2. Our proof of evaluation binding is directly taken from Kate et al.'s reduction to q-SDH. However, we also require that the scheme is bounded polynomial

Common input: $\mathsf{info} = bp, \mathtt{srs}, \mathsf{max}$
Prover's input: $f(X)$

$\underline{\mathsf{Commit}(\mathsf{info}, f(X)) \mapsto F}$:
$F \leftarrow g^{\alpha x^{d-\mathsf{max}} f(x)}$
return F

$\underline{\mathsf{Open}(\mathsf{info}, F, z, f(X)) \mapsto (f(z), W)}$:
$w(X) \leftarrow \frac{f(X)-f(z)}{X-z}$
$W \leftarrow g^{w(x)}$
return $(f(z), W)$

$\underline{\mathsf{pcV}(\mathsf{info}, F, z, (v, W)) \mapsto 0/1}$:
check $e(W, h^{\alpha x})e(g^v W^{-z}, h^{\alpha}) = e(F, h^{x^{-d+\mathsf{max}}})$
return 1 if all check passes, else return 0

Figure 6.3: Polynomial commitment scheme inspired by Kate et al [KZG10].

extractable; i.e., any algebraic adversary that opens a commitment F knows an opening $f(X)$ with powers $-d \leq i \leq \mathsf{max}, i \neq 0$. Kate et al. prove only that their scheme is "strongly correct"; i.e., if an adversary knows an opening $f(X)$ with polynomial degree to a commitment then $f(X)$ has degree bounded by d. In this sense Kate et al. are implicitly relying on a knowledge assumption, because there is no guarantee that an adversary that can open a commitment knows a polynomial inside the commitment. We prove our adapted polynomial commitment scheme secure in the algebraic group model and this proof may be of independent interest.

Our proof uses the fact that $f(X) - f(z)$ is divisible by $(X - z)$, even for Laurent polynomials. To see this observe that

$$\begin{aligned} f(X) - f(z) &= \sum_{-d}^{d} a_i X^i - a_i z^i \\ &= \sum_{i=1}^{d} a_i (X - z)(X^{i-1} + zX^{i-2} + \ldots z^{i-1}) + 0a_0 \\ &+ \sum_{i=-1}^{-d} a_i (X - z)(-z^{-1}X^{-i} - z^{-2}X^{-i+1} - \ldots - z^{-i}X^{-1}) \end{aligned}$$

Theorem 14. *In the algebraic group model, the polynomial commitment scheme in Figure 6.3 is evaluation binding and bounded polynomial extractable under the* $2d$-DLOG *assumption.*

Proof. We closely follow the structure used by Fuchsbauer et al. [FKL18, Theorem 7.2]. We consider an algebraic adversary $\mathcal{A}_{\mathsf{alg}}$ against the security of the polynomial commitment scheme; by definition, this means that $\mathcal{A}_{\mathsf{alg}}$ breaks either bounded polynomial extractability or evaluation binding; i.e., that

$$\mathcal{A}^{\mathsf{pc}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{\mathsf{extract}}_{bp,\mathcal{A}_{\mathsf{alg}}} + \mathcal{A}^{\mathsf{bind}}_{bp,\mathcal{A}_{\mathsf{alg}}}.$$

We show that

$$\mathcal{A}^{\mathsf{pc}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{B}_{\mathsf{alg}}} + \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{C}_{\mathsf{alg}}}$$

for adversaries $\mathcal{B}_{\mathsf{alg}}$ and $\mathcal{C}_{\mathsf{alg}}$, which proves the theorem.

We start with bounded polynomial extractability, where we show that

$$\mathcal{A}^{\mathsf{extract}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{B}_{\mathsf{alg}}}.$$

An adversary $\mathcal{B}_{\mathsf{alg}}(g^1, g^x, \ldots, g^{x^q})$ simulates the bounded polynomial extractability game with $\mathcal{A}_{\mathsf{alg}}$ as follows.

1. When $\mathcal{A}_{\mathsf{alg}}$ queries its oracle $\mathsf{U}\text{-}\mathcal{O}_{\mathsf{s}}$ on setup, $\mathcal{B}_{\mathsf{alg}}$ chooses random values (u_1, u_2) and uses its DLOG instance to generate and return an SRS with implicit randomness $(u_1 x, u_2 x)$.

2. When $\mathcal{A}_{\mathsf{alg}}$ queries its oracle on update, $\mathcal{B}_{\mathsf{alg}}$ uses the algebraic representation provided by $\mathcal{A}_{\mathsf{alg}}$ to learn the randomness (x_i, α_i) used by $\mathcal{A}_{\mathsf{alg}}$ in generating its intermediate SRSs (if any exist). It then picks new randomness (u'_1, u'_2) and updates its own stored randomness as $(u_1, u_2) = (x_i u'_1 u_1, \alpha_i u'_2 u_2)$. It then uses this randomness (consisting of its old randomness, the randomness of $\mathcal{A}_{\mathsf{alg}}$, and its new randomness) to simulate the update proof. It returns the simulated update proof and the new SRS to $\mathcal{A}$.

3. When $\mathcal{A}_{\mathsf{alg}}$ queries its oracle on final, $\mathcal{B}_{\mathsf{alg}}$ behaves as the honest oracle.

4. $\mathcal{B}_{\mathsf{alg}}$ runs $(F, z, v, W) \xleftarrow{r} \mathcal{A}_{\mathsf{alg}}(bp, \mathtt{srs}, \mathsf{max})$.

5. The randomness r determines multivariate polynomials

$$f(X, X_\alpha) = f_x(X) + X_\alpha f_\alpha(X),$$
$$w(X, X_\alpha) = w_x(X) + X_\alpha w_\alpha(X),$$

   such that

$$F = g^{f(xu_1, xu_2)} \text{ and } W = g^{w(xu_1, xu_2)}.$$

   From these polynomials, $\mathcal{B}_{\mathsf{alg}}$ computes the polynomial

$$Q_1(X, X_\alpha) = X_\alpha(X - z)w(X, X_\alpha) + vX_\alpha - X^{-d+\mathsf{max}} f(X, X_\alpha).$$

   It aborts if $Q_1(X, X_\alpha) = 0$.

6. Define the univariate polynomial $Q'_1(X) = Q_1(u_1 X, u_2 X)$. $\mathcal{B}_{\mathsf{alg}}$ aborts if $Q'_1(X) = 0$.

7. $\mathcal{B}_{\mathsf{alg}}$ factors $Q'_1(X)$ to obtain its roots (of which there are at most $4d$) and checks them against the q-DLOG instance to determine if x is among them. If so, it returns x. Otherwise it returns $\perp$.

Now let us analyze the probability that $\mathcal{A}_{\mathsf{alg}}$ breaks bounded polynomial extractability; i.e., that

$$f(X, X_\alpha) \neq X_\alpha X^{d-\mathsf{max}} \left(\sum_{i=-d, i\neq 0}^{\mathsf{max}} a_i X^i \right),$$

but that $\mathcal{B}_{\mathsf{alg}}$ does not return the target x. This happens if (1) $\mathcal{B}_{\mathsf{alg}}$ aborts in Step 5, (2) $\mathcal{B}_{\mathsf{alg}}$ aborts in Step 6, or (3) if x is not amongst the roots obtained in Step 7. We consider these three scenarios in turn.

In Step 5, if $Q_1(X, X_\alpha) = 0$ then

$$X_\alpha(X - z)w(X, X_\alpha) + vX_\alpha - (X^{-d+\mathsf{max}}) f(X, X_\alpha) = 0$$

which implies that

$$(X - z)w_x(X) + v - (X^{-d+\mathsf{max}}) f_\alpha(X) = 0$$

and $(X - z)$ divides $(X^{-d+\mathsf{max}})f_\alpha(X) - v$ and $f_\alpha(X)$ has non-zero terms between $-\mathsf{max}$ and d. Thus $f_\alpha(X)$ has no terms with degree less than $-\mathsf{max}$. Moreover $f_\alpha(X)$ has no zero term because this is not given in the reference string. Thus $\mathcal{B}$ aborts in this step only if $f(X, X_\alpha)$ is as assumed, which means $\mathcal{A}_{\mathsf{alg}}$ has not broken bounded polynomial extractability.

In Step 6, $\mathcal{B}_{\mathsf{alg}}$ aborts only if $Q_1(u_1X, u_2X) = 0$. By the Schwartz-Zippel lemma, the probability of this occurring is bounded by $\frac{(4d)^2}{p-1}$ where d is the total degree of Q (recall we have negative powers). Following the generic bound for Boneh and Boyen's SDH assumption [BB08] we may assume that $\mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{B}_{\mathsf{alg}}} \geq \frac{q^2}{p-1}$; i.e., that the probability that $\mathcal{B}_{\mathsf{alg}}$ aborts in this way is negligible.

In Step 7, $Q_1(u_1x, u_2x)$ exactly defines the verifier's equation, so if $\mathcal{A}_{\mathsf{alg}}$ succeeds then $Q_1(u_1x, u_2x) = 0$. Thus $Q_1'(x) = 0$ and x is a root of $Q_1'(X)$.

Thus when $\mathcal{A}_{alg}$ succeeds at breaking bounded polynomial extractability, $\mathcal{B}_{\mathsf{alg}}$ returns x unless $Q_1(u_1X, u_2X) = 0$, which happens with bounded probability. Thus

$$\mathcal{A}^{\mathsf{extract}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{B}_{\mathsf{alg}}}$$

as desired.

We now consider evaluation binding, where we show that

$$\mathcal{A}^{\mathsf{bind}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{C}_{\mathsf{alg}}}.$$

In fact, $\mathcal{C}_{\mathsf{alg}}$ does not act directly on the q-DLOG assumption, but rather on the q-SDH assumption [BB08], which states that given $(g, g^x, \ldots, g^{x^q})$ it is hard to compute $(c, g^{\frac{1}{x-c}})$ for some value c. In particular we show that if $\mathcal{A}_{\mathsf{alg}}$ can open their commitment at z to two different evaluations then $\mathcal{C}_{\mathsf{alg}}$ can compute a tuple of this form. Following the generic bound for q-SDH [BB08], this assumption is implied by q-DLOG so the result holds.

The adversary $\mathcal{C}_{\mathsf{alg}}(g^1, g^x, \ldots, g^{x^q})$ simulates the evaluation binding game with $\mathcal{A}_{\mathsf{alg}}$ as follows.

1. $\mathcal{C}_{\mathsf{alg}}$ behaves just as $\mathcal{B}_{\mathsf{alg}}$ did in its Steps 1-4 in answering oracle queries.
2. $\mathcal{C}_{\mathsf{alg}}$ runs $(F, z, v_1, v_2, W_1, W_2) \xleftarrow{r} \mathcal{A}_{\mathsf{alg}}(bp, \mathtt{srs}, \mathsf{max})$.
3. If $v_1 \neq v_2$ $\mathcal{C}_{\mathsf{alg}}$ returns $(z, (W_1W_2^{-1})^{\frac{1}{v_2-v_1}})$. Otherwise it returns $\perp$.

If $v_1 \neq v_2$ then

$$e(W, h^\alpha)e(W^{-z}g^v, h^\alpha) = e(W, h^\alpha)e(W'^{-z}g^{v'}, h^\alpha)$$

and rearrangement yields

$$e(WW'^{-1}, h^{\alpha(x-z)}) = e(g^{v'-v}, h^\alpha).$$

Thus $\mathcal{C}_{\mathsf{alg}}$ returns $(z, g^{\frac{1}{x-z}})$ and

$$\mathcal{A}^{\mathsf{bind}}_{bp,\mathcal{A}_{\mathsf{alg}}} \leq \mathcal{A}^{q\text{-}\mathsf{DLOG}}_{bp,\mathcal{C}_{\mathsf{alg}}}$$

as required. □

6.6 Succinct Signatures of Correct Computation

In Section 6.5, we provided our main Sonic construction assuming a secure polynomial commitment scheme and signature of correct computation. While we showed a secure polynomial commitment scheme in Section 6.5.2, it remains to provide an instantiation of a secure signature of correct computation $(\mathsf{scP}, \mathsf{scV})$ [PST13]. Recall from Section 6.5 that Sonic uses a signature of correct computation to ensure that an element s is equal to $s(z, y)$ for a known polynomial

$$s(X, Y) = \sum_{i,j=-d}^{d} s_{i,j}X^iY^j.$$

We require the soundness notion that no adversary can convince an scV verifier unless $s = s(z, y)$, and as usual require this property to hold even against adversaries that can update the SRS. We provide two competing realizations of signatures of correct computation. The first one is described in this section and it is calculated by a prover, and has succinct size and verifier computation. The second one considers settings in which one can use untrusted helpers to improve practical efficiency, and we describe it in Section 6.7.

We use the structure of $s(X, Y)$ in order to prove its correct calculation using a *permutation argument*, which itself has a *grand-product argument* as an underlying component. We take inspiration from our main construction and from the permutation and grand-product arguments described by Bayer and Groth [BG12] and by Bootle et al [BCG+17]. We restrict ourselves to constraint systems for which $s(X, Y)$ can be expressed as the sum of M polynomials, where the j-th such polynomial is of the form

$$\Psi_j(X, Y) = \sum_{i=1}^{n} \psi_{j,\sigma_{j,i}} X^i Y^{\sigma_{j,i}}$$

for (fixed) polynomial permutation $\boldsymbol{\sigma}_j$ and coefficients $\psi_{j,i} \in \mathbb{F}$. By introducing additional multiplication constraints to replace any linear constraints that do not fit this format, we can coerce any constraint system in Section 6.4 into the correct form.

To expand further, our constraint system is determined by vectors $\boldsymbol{u_q}, \boldsymbol{v_q}, \boldsymbol{w_q}$ of size n that are typically sparse. To represent Ψ_j in the desired form, we require that each power of Y in $s(X, Y)$ appears in no more than M occurrences, which means that for all $1 \leq i \leq n$, only three values of $\boldsymbol{u_q}, \boldsymbol{v_q}, \boldsymbol{w_q}$ can be non-zero. If $\boldsymbol{u_q}$ is too dense (the maximum density is determined by the number of permutation arguments and there is an efficiency trade off between proof size and prover computation), we split our original constraint into two or more constraints: we set $0 = \sum_{i=1}^{n-\ell} a_i u_{q,i} - a_{n+1}$ and

$$k_q = a_{n+1} + \sum_{i=n-\ell+1}^{n} a_i u_{q,i} + \boldsymbol{b} \cdot \boldsymbol{v_q} + \boldsymbol{c} \cdot \boldsymbol{w_q}.$$

In doing so we have extended the length of $\boldsymbol{a}$ by one, and so also must extend the length of $\boldsymbol{b}$ and $\boldsymbol{c}$ by one to obtain a dummy multiplicative constraint. The precise number of additional multiplication constraints depends on the number of additive constraints (essentially it implies that if there are more than $2n$ addition constraints in an arithmetic circuit, then these are no longer free). In practice we found that the increase in the number of multiplication constraints for SHA256 circuits is approximately a factor of 3 when $M = 3$.

Our signature of correct computation uses a polynomial permutation argument, which itself uses a grand-product argument. The permutation argument allows us to verify that each polynomial commitment contains $\Psi_j(X, y)$, and this can then be opened at z to verify that $\Psi_j(z, y)$ has been calculated correctly. The purpose of this argument is to offload the verifier's computational costs onto the prover. After using batching techniques described in the full version [MBKM], we get proof sizes of approximately 1kB.

The permutation verifier does not take in the permutation itself, but a derived reference string $\mathtt{srs}_\Psi$ that can be deterministically generated from the global $\mathtt{srs}$ and the permutation Ψ using 4 multi-exponentiations of size n in $\mathbb{G}_1$. The cost of generating the derived reference string is then amortized when the protocol is run over multiple instances.

6.6.1 Polynomial Permutation Argument

A polynomial permutation argument is defined by three DPT protocols

- $\mathtt{srs}_\Psi \leftarrow \mathsf{Derive}(bp, \mathtt{srs}, \Psi(X, Y))$ takes as input a bilinear group, a structured reference string, and a polynomial $\Psi(X, Y) = \sum_{i=1}^{n} \psi_{\sigma_i} X^i Y^{\sigma_i}$. It outputs a derived reference string $\mathtt{srs}_\Psi$.
- $(\psi, \mathsf{perm}) \leftarrow \mathsf{permP}(\mathtt{srs}_\Psi, y, z, \Psi(X, Y))$ takes as input a derived reference string, two points in the field, and a polynomial $\Psi(X, Y)$. It outputs $\psi = \Psi(z, y)$ and a proof perm.

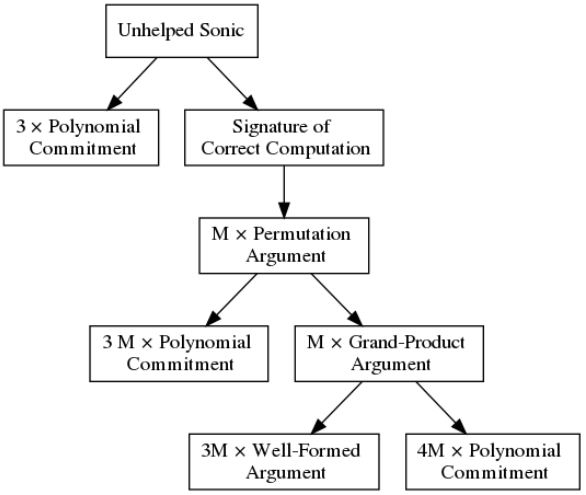

Figure 6.4: Sonic is built using a polynomial commitment scheme and a signature of correct computation. Here we describe how the prover can construct the signature of correct computation using permutation arguments, grand-product arguments, and the polynomial commitment scheme described in Section 6.5.2.

- $0/1 \leftarrow \mathsf{permV}(\mathtt{srs}_\Psi, y, z, (\psi, \mathsf{perm}))$ takes as input a derived reference string, two points in the field, an evaluation, and a proof. It outputs a bit indicating acceptance ($b = 1$), or rejection ($b = 0$).

We require that this scheme is sound; i.e., an adversary can convince a verifier only if $\psi = \Psi(z, y)$. As with our earlier building blocks, we require this to hold even against adversaries that can update the SRS. Our polynomial permutation argument is given in the full version [MBKM].

Theorem 15. *The signature of computation scheme in Figure 6.5 is sound when instantiated using a sound permutation argument.*

Proof. The polynomial $s(X, Y)$ is given by $\sum_j \psi_j(X, Y)$. The soundness of the permutation argument gives us that no adversary can convince the verifier of Ψ_j unless ψ_j is the correct evaluation of Ψ_j at (z, y); i.e., $\psi_j = \sum_{i=1}^{n} \psi_{j,i} z^i y^{\sigma_{j,i}}$. Thus the verifier is convinced if and only if $s = \sum_j \psi_j$ is the correct evaluation of $s(X, Y)$ at z, y. □

6.6.2 Grand-Product Argument

One of the main components of our polynomial permutation argument is a grand-product argument. A grand-product argument is defined by two DPT protocols

- $\mathsf{gprod} \leftarrow \mathsf{gprodP}(bp, \mathtt{srs}, A, B, a(X), b(X))$ takes as input the bilinear group, the SRS, two polynomial commitments, and two openings such that $\prod_i a_i = \prod_i b_i$.
- $0/1 \leftarrow \mathsf{permV}(bp, \mathtt{srs}, A, B, \mathsf{gprod})$ takes as input the bilinear group, the SRS, two polynomial commitments, and a proof. It outputs a bit indicating acceptance ($b = 1$), or rejection ($b = 0$).

We require that this scheme is knowledge-sound; i.e., an adversary can convince a verifier only if it knows openings to A and B whose coefficients have the same grand-product; i.e., such that $\prod_i a_i = \prod_i b_i$. Again, we require this to hold even against adversaries that can update the SRS. Our grand-product argument is given in the full version [MBKM].

Common input: $\mathsf{info} = \{\mathtt{srs}_{\Psi_j}\}_{j=1}^{M}, y, z$

$\underline{\mathsf{scP}(\mathsf{info}, \{\Psi_j(X,Y)\}_{j=1}^{M}) \mapsto (s, \{\psi_j, \mathsf{perm}_j\}_{j=1}^{M}):}$
for $1 \leq j \leq M$:
  $(\psi_j, \mathsf{perm}_j) \leftarrow \mathsf{permP}(\mathtt{srs}_{\Psi_j}, y, z, \Psi_j(X,Y))$
$s \leftarrow \sum_{j=1}^{M} \psi_j$
return $(s, \{\psi_j, \mathsf{perm}_j\}_{j=1}^{M})$

$\underline{\mathsf{scV}(\mathsf{info}, (s, \{\psi_j, \mathsf{perm}_j\}_{j=1}^{M})) \mapsto 0/1:}$
check $s = \sum_{j=1}^{M} \psi_j$
for $1 \leq j \leq M$:
  check $\mathsf{permV}(\mathtt{srs}_{\Psi_j}, y, z, (\psi_j, \mathsf{perm}_j))$
return 1 if all checks pass, else return 0

Figure 6.5: A signature of correct computation using a permutation argument.

	Helper	Verifier	Proof size
Helped	$\mathcal{O}(mn\log(n))$	$\mathcal{O}(m) + \mathcal{O}(n)$	$3m+3\ \mathbb{G}_1$, $2m+1\ \mathbb{F}$
Unhelped	-	$\mathcal{O}(m)$	$16m\ \mathbb{G}_1$, $14m\ \mathbb{F}$

Table 6.1: Computational efficiency and proof size for the sc with respect to the helped verifier. Here n is the number of multiplication gates and m is the number of proofs for the same constraint system. Although the unhelped version has better asymptotic efficiency, in practice the helped verifier is more efficient.

6.7 Signatures of Correct Computation with Efficient Helped Verification

Recall that Sonic uses a signature of correct computation to ensure that an element s is equal to $s(z, y)$ for a known polynomial

$$s(X,Y) = \sum_{i,j=-d}^{d} s_{i,j} X^i Y^j.$$

In Section 6.6 we described a signature of correct computation that is calculated directly by a prover, and has succinct size and verifier computation. Alternatively, in some settings one can use untrusted helpers to improve practical efficiency, which we describe in this section. In the helper setting, proof sizes and prover computation are significantly more efficient.

In the amortized setting, where one is proving the same thing many times, we can use "helpers" in order to aggregate many signatures of correct computation at the same time. The proofs provided by the helper are succinct and the helper can be run by anyone (i.e., they do not need any secret information from the prover). Verification requires a one-off linear-sized polynomial evaluation in the field and an addition two pairing equations per proof. Compared to the unhelped costs (which require an additional 4 pairings per proof) this is more efficient assuming there is a sufficiently large number of proofs in the batch. As discussed in the introduction, the natural candidate for this role in the setting of blockchains is a miner, as they are already investing computational energy into the system. An efficiency overview is given in Table 6.1.

The algorithm for our helped signature of correct computation is given in Figure 6.7. The helper is denoted by hscP and the verifier is denoted by hscV. Roughly the idea is as follows. The helper commits to $s(X, y_j)$ for each element y_j. The verifier provides a random challenge u. The helper commits to $s(u, X)$, and then opens its commitment to $s(X, y_j)$ at u and its commitment $s(u, X)$ at y_j and checks the two are equal. The verifier

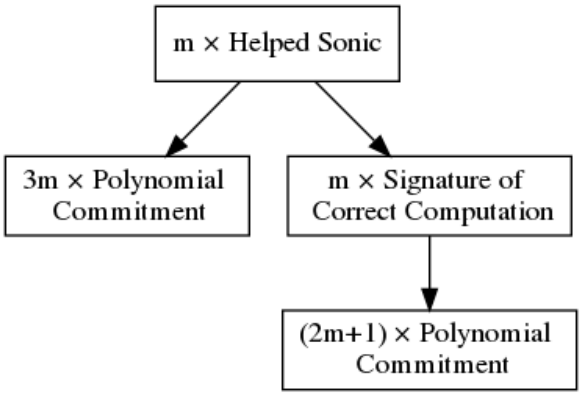

Figure 6.6: Sonic can be constructed using a signature of correct computation that is calculated by a helper as opposed to directly by the prover. The helper algorithm is run on a batch of proofs, and provides the setting in which Sonic obtains the best practical efficiency.

provides a random challenge v. The helper opens $s(u, X)$ at v. The verifier computes $s(u, v)$ for itself and checks that the helper's opening is correct.

Theorem 16. *The aggregated signature of correct computation in Figure 6.7 is sound when instantiated using a secure polynomial commitment scheme.*

Proof. Bounded polynomial extraction of the underlying polynomial commitment gives us that there exist algebraic extractors that output degree-d Laurent polynomials $s'_j(X)$ and $c'(X)$ such that $S_j = g^{\alpha s'_j(x)}$ and $C = g^{\alpha c'(x)}$. First observe that the probability that $c'(v) = s(u, v)$ at a randomly chosen v but that $c'(X) \neq s(u, X)$ is negligible in a sufficiently large field. Second observe that given $c'(X) = s(u, X)$, a PPT algebraic adversary can open C only at a (not randomly chosen) value y_j to $s(u, y_j)$. Finally observe that the probability that $s'_j(X) = s(u, y_j)$ at a randomly chosen u but that $s'_j(X) \neq s(X, y_j)$ is negligible in a sufficiently large field. Thus soundness follows from the evaluation binding of the polynomial commitment. □

6.8 Implementation

In order to compare the concrete performance of our construction to other protocols we provide an open-source implementation in Rust [Son] of Sonic implemented with helpers. We chose to implement only this variant of Sonic because it has better practical efficiency. The numbers in Table 6.2 were obtained on CPU i7 2600K with 32 GB of RAM, running at 3.4 GHz.

In terms of our parameters, we make use of the BLS12-381 elliptic curve construction, which is designed so that its group order is a prime p such that $\mathbb{F}_p$ is equipped with large 2^n roots of unity for performing fast polynomial multiplications with radix-2 fast-Fourier transforms. BLS12-381 targets the 128-bit security level. Kim and Barbulescu [KB16a] describe an optimization to the Number Field Sieve algorithm, analyzed further by Babalescu and Duquesne [KB16b], which may reduce security to 117 bits, but the attack requires a (currently unknown) efficient algorithm for scanning a large space of polynomials.

Proof verification is dominated by a set of pairing equation checks and an evaluation of $s(X, Y)$ in the scalar field. Most of the pairings within (and amongst many) proof verifications involve fixed elements in $\mathbb{G}_2$, so the verifier can combine all of them into a single equation with a probabilistic check. In the context of batch verification each individual proof thus requires arithmetic only in $\mathbb{G}_1$. Only a small, fixed number of pairing operations are performed at the end.

As mentioned in Section 6.7, the evaluation of $s(X, Y)$ can be done once for a batch of proofs given some post-processing by an untrusted helper. We consider the performance of batch verification with this post-processing.

Input size (bits)	Gates	Size		Timing			
		SRS (MB)	Proof (bytes)	SRS (s)	Prove (s)	Helper (s)	Helped Verifier (ms)
Pedersen hash preimage (input size)							
48	203	0.47	256	2.24	0.15	0.09	0.69
384	1562	3.74	256	17.62	0.84	0.46	0.72
Unpadded SHA256 preimage							
512	39516	91.05	256	422.39	14.63	8.41	0.68
1024	78263	182.09	256	831.87	28.93	14.23	0.68
1536	117010	273.14	256	1301.43	38.86	21.54	0.68

Table 6.2: Sonic's efficiency in proving knowledge of x such that $H(x) = y$ for different sizes of x. Numbers are given to two significant digits. The first rows are for the Pedersen hash function and the final rows are for SHA256. "Helper" and "Helped Verifier" are the marginal cost of aggregating and verifying an additional proof assuming that the helper has been run. These are calculated by batch-verifying 100 proofs, subtracting the cost to verify one, and dividing by 99.

In each individual proof we must compute $k(y)$ depending on our instance. We keep this polynomial sparse by having coefficients only in our instance variables, and keeping all other coefficients zero. If constants are needed in the circuit, they are expressed with coefficients of an instance variable that is fixed to one.

We provide an adaptor which translates circuits written in the form of quadratic "rank-1 constraint systems" (R1CS) [BCG$^+$13], a widely deployed NP language currently undergoing standardization, into the system of constraints natural to our proving system. This adds some constant amount of overhead during proving and verifying steps, but eases implementation and comparison with existing constructions.

The numbers obtained are relevant only to batched proofs, so we wrote an idealized verifier of the Groth 2016 scheme [Gro16], where a batch of proofs are verified together. In this idealized version we assume the $\mathbb{G}_2$ elements do not need to be deserialised and that there is only one public input. We found the marginal cost of verification was around 0.6ms, compared to Sonic's 0.7ms. We thus claim that Sonic has verification time which is competitive with the state-of-the-art for zk-SNARKs, but unlike prior zk-SNARKs has a universal and updatable SRS.

In Table 6.2 we mimicked Bulletproofs [BBB$^+$18, Table 3] in measuring the results of our Sonic implementation. Our implementation is not constant time, however, which may affect this comparison (or indeed the comparison of prover performance to any implementation with constant-time algorithms). We measured the efficiency of the prover, the verifier, and the helped verifier in proving knowledge of x such that $H(x) = y$. Proof sizes are always 256 bytes and verifier computation is always around 0.7ms. In Bulletproofs, in contrast, the proof size for the unpadded 512-bit SHA256 preimage is 1376 bytes and verification time is 41.52 ms, although as we mention this comparison is not exact give in particular that their system was throttled to 2 GHz and that there are optimized implementations for fixed circuits.[2] The runtime of our prover goes up in a roughly linear fashion, as expected. The cost of the helped verifier, in contrast, remains the same for all circuit sizes.

6.9 Relation to Distributed Ledgers

Zero-knowledge protocols have gained significant traction in recent years in the application domain of distributed ledgers, which has led to the development of new protocols with significant performance gains. At the same time, the requirements of this application have given rise to protocols with new features, such as an untrusted setup

[2] `https://github.com/dalek-cryptography/zkp`

and a reference string that allows one to prove more than a single relation. We present Sonic, which captures a valuable set of tradeoffs between these key functional requirements of untrusted setup and universality. At the same time, as we demonstrate via a prototype implementation, Sonic has proof sizes and verification time that are competitive with the state-of-the-art.

Common input: $\mathsf{info} = bp, \mathtt{srs}, \{z_j, y_j\}_{j=1}^m, s(X, Y)$

$\underline{\mathsf{hscP}_1(\mathsf{info}) \mapsto (\{S_j, s_j, W_j\}_{j=1}^m)}$:
for $1 \leq j \leq m$:
  $S_j \leftarrow \mathsf{Commit}(bp, \mathtt{srs}, d, s(X, y_j))$
  $(s_j, W_j) \leftarrow \mathsf{Open}(S_j, z_j, s(X, y_j))$
send $\{S_j, s_j, W_j\}_{j=1}^m$

$\underline{\mathsf{hscV}_1(\mathsf{info}, \{S_j, s_j, W_j\}_{j=1}^m) \mapsto u}$:
send $u \xleftarrow{\$} \mathbb{F}_p$

$\underline{\mathsf{hscP}_2(u) \mapsto \{\hat{s}_j, \hat{W}_j, Q_j\}_{j=1}^m}$:
$C \leftarrow \mathsf{Commit}(bp, \mathtt{srs}, d, s(u, X))$
for $1 \leq j \leq m$:
  $(\hat{s}_j, \hat{W}_j) \leftarrow \mathsf{Open}(S_j, u, s(X, y_j))$
  $(\hat{s}_j, Q_j) \leftarrow \mathsf{Open}(C, y_j, s(u, X))$
send $\{\hat{s}_j, \hat{W}_j, Q_j\}_{j=1}^m$

$\underline{\mathsf{hscV}_2(\{\hat{s}_j, \hat{W}_j, Q_j\}_{j=1}^m) \mapsto v}$:
send $v \xleftarrow{\$} \mathbb{F}_p$

$\underline{\mathsf{hscP}_2(v) \mapsto Q_v}$:
$(s(u, v), Q_v) \leftarrow \mathsf{Open}(C, v, s(u, X))$
send Q_v

$\underline{\mathsf{hscV}(Q_v) \mapsto 0/1}$:
$s_v \leftarrow s(u, v)$
for $1 \leq j \leq m$:
  check $\mathsf{pcV}(bp, \mathtt{srs}, S_j, d, z_j, (s_j, W_j))$
  check $\mathsf{pcV}(bp, \mathtt{srs}, S_j, d, u, (\hat{s}_j, \hat{W}_j))$
  check $\mathsf{pcV}(bp, \mathtt{srs}, C, d, y_j, (\hat{s}_j, Q_j))$

check $\mathsf{pcV}(bp, \mathtt{srs}, C, d, v, (s_v, Q_v))$
return 1 if all checks pass, else return 0

Figure 6.7: The helper protocol for computing aggregated signatures of correct computation.

Chapter 7

Secure Groups and Their Applications in MPC-based Threshold Cryptography

7.1 Summary

We propose a scheme to implement finite groups as oblivious data structures, meaning that no information can be inferred about the values of the group elements after a sequence of operations. For a given group, the scheme defines the oblivious representation of group elements and oblivious operations on group elements. Operations include the group law, exponentiation and inversion, random sampling and encoding/decoding.

The oblivious operations are defined by a set of secure multiparty computation (MPC) protocols. We demonstrate these protocols in a standard setting for information theoretically secure MPC, tolerating a dishonest minority of passively corrupt parties. Practical protocols are presented for the group of quadratic residues, elliptic curves groups and class groups of imaginary quadratic order. The Python package for secure groups will be published with the final version of this work.

To illustrate the application of secure groups, we extend a classical threshold cryptosystem with protocols to simplify in- and output to a multiparty computation.

7.2 Introduction

The cryptographic scheme for *secure groups* introduced in this chapter aims to significantly simplify secure computation involving finite groups, in particular for applications in threshold cryptography. With secure groups, a protocol engineer can implement secure group operations by composing time-tested MPC protocols.

Recall that a finite group is a finite set that satisfies closure, associativity, identity and invertibility. The secure group scheme implements finite groups in a privacy-preserving manner: Both the representation of group elements and the implementation of the group operations are oblivious, meaning that no information can be inferred about the values of the group elements unless they are public. The scheme includes necessary protocols and algorithms to instantiate this oblivious data structure in the MPC setting, notably secure exponentiation, random sampling and encoding/decoding. We distinguish between general finite groups and groups commonly used in cryptography, particularly elliptic curve groups and class groups.

For general groups, if a faithful linear representation over a finite field is available from modular representation theory, it directly permits an oblivious representation of the group elements (as square matrices over a finite field) and an oblivious implementation of the group operation (as matrix multiplication).

Cryptographic applications often involve finite cyclic groups. While finding the linear representation for cyclic groups is trivial, additional requirements make their secure representation non-trivial in the MPC setting.

First, applications typically require hardness of the discrete logarithm problem for the particular group, making linear representations unsuitable. Second, implementing the group operation in an oblivious way (including

inversion, squaring and exponentiation) is not trivial for general finite groups. We present practical examples for quadratic residues, elliptic curve groups and class groups.

7.2.1 Contributions

The Concept of Secure Groups. We introduce secure groups as cryptographic schemes to facilitate concrete implementations of finite groups in MPC. Implementing finite groups in an MPC setting minimally requires an oblivious representation of group elements and an oblivious implementation of the group operation. We note that Bar-Ilan and Beaver already considered a generic protocol for secure inversion of group elements, as a natural generalization of secure matrix multiplication [BIB89, Lemma 6].

In this chapter we study oblivious group representations and operations in detail, which leads to a surprising number of interesting research questions such as: How to obliviously sample a random element from a group? Or, how to define an encoding that permits an efficient oblivious decoding of a secret-shared group element? This work is the first comprehensive study of such questions.

Oblivious Group Representation and Operation. We define practical oblivious representations and operations for a set of frequently used finite groups. Depending on the type of group, we present different insights:

- The multiplicative group $\mathbb{F}_q^*$ and any of its subgroups directly permit an oblivious representation and group operation using secret shares over $\mathbb{F}_q$. Encoding and decoding from and to the secure group is not trivial, however. We present several techniques and their trade-offs.
- Groups defined on elliptic curves over finite fields directly permit an oblivious representation. To facilitate the implementation of such groups, we employ the *complete* formulas known for Weierstrass and Edwards curves.[1] We highlight one result from [HWCD08] for parallel architectures, which yields an oblivious group law with low multiplicative depth.
- Class groups of positive definite binary quadratic forms are studied in more depth. For class groups with a given discriminant Δ, we study the representation of elements using integer shares defined over finite fields. We define the secure group operation by introducing a protocol to obliviously compute the extended gcd of two secret-shared integers. We also discuss random sampling, encoding and exponentiation for secure class groups, as they present a case example of such techniques for groups of unknown order.

Extended GCD Protocol. For form class groups of imaginary quadratic order, the group operation called *composition* requires computation of the extended gcd. To calculate the extended gcd of two secret-shared integers, we have developed an efficient secure xgcd protocol in draft. The final protocol will be published with PRIViLEDGE deliverable D2.4.

Secure Group Protocols. The protocols for (secure) encoding/decoding, random sampling, inversion and exponentiation for finite groups of known and unknown order aim to optimize both generality and efficiency.

For example, we suggest a generic, probabilistic encoding technique based on [Kob87, Section 3] that permits efficient decoding of a secret group element. Our technique to sample elements for generic groups is based on the notion of random walks in groups [Dia88, ACS02, ER65]. A particularly efficient technique by Dixon [Dix08] is adopted, which in turn permits us to apply a random self-reduction technique to construct efficient secure exponentiation protocols for finite groups.

[1] Edwards [Edw07] introduced a new normal form of elliptic curves with group law formulas that can be stated explicitly and do not have exceptional cases. Bernstein and Lange [BL07] introduced fast complete formulas for addition and doubling in projective coordinates. Hisil et al. [HWCD08] improved complete formulas for twisted Edwards curves, in particular one that parallelizes multiplications. Renes et al. [RCB16] presented complete formulas for prime order elliptic curves, originally from Bosma and Lenstra [BL95], optimized for cryptographic applications.

Extending Threshold Cryptosystems and Proof Systems. To demonstrate secure groups, we extend the classical threshold cryptosystem [Ped91a] with MPC to gain additional functionalities: One protocol to threshold decrypt ciphertexts to Shamir shares and a second proxy reencryption protocol that converts a ciphertext for one public key to a ciphertext for a second public key. Examples of recent work on threshold cryptosystems using MPC are [SA19] and [GG19].

Furthermore, secure groups make it straightforward to conduct the prover side of a Σ-protocol or succinct non-interactive argument of knowledge (SNARK) in MPC. We illustrate this by constructing the proof of a Pinocchio SNARK [PHGR13] via a secure multi-party computation using our library. This, in turn, allows convenient constructions of *verifiable MPC* protocols, i.e. proof systems that enable MPC parties to create a publicly verifiable proof of correctness of the MPC computation, even in the extreme case that all MPC parties are corrupt [SVdV16]. These examples can be found in the Python package published with the final version of this work.

7.2.2 Roadmap

The chapter is organized as follows. Section 7.3 introduces preliminaries for finite groups and secure multiparty computation. Section 7.4 introduces secure groups and presents constructions for quadratic residues, elliptic curve groups and class groups. Section 7.5 presents protocols to implement secure groups: encoding/decoding, generating random elements and secure exponentiation. Section 7.6 presents a threshold cryptosystem built from secure groups. Section 7.7 concludes the chapter by introducing our Python package for secure groups.

7.3 Preliminaries

7.3.1 Finite Groups

Throughout, $\mathbb{G}$ denotes a finite group, $\mathbb{G}_n$ denotes a group of order n, and $\mathbb{Z}_n$ is shorthand for $\mathbb{Z}/n\mathbb{Z}$ unless explicitly stated otherwise. Group notation is multiplicative, if not mentioned explicitly otherwise. A *representation* ϱ refers to a map associating a group element with a tuple over (multiple) finite fields. If we refer to representations as *linear transformations of vector spaces* as per representation theory, we will indicate this explicitly, denote the representation by ρ and use the term *linear representation*.

7.3.2 Prime-Order Subgroups of $\mathbb{F}_q^*$

Let g be an element of prime order p in $\mathbb{F}_q^*$. Then $p \mid q-1$ and $\mathbb{G} = \langle g \rangle$ is the subgroup of order p in $\mathbb{F}_q^*$. If $p = (q-1)/2$, then $\mathbb{G}$ is the Quadratic Residue group (QR group). Often q is prime (not a prime power), then $\mathbb{F}_q = \mathbb{Z}_q$.

7.3.3 Elliptic Curve Groups

Let $E(\mathbb{F}_q)$ denote the elliptic-curve group of E. Let $1_{\mathbb{G}}$ denote the *identity* element. Cryptographic purposes require that the cyclic (sub)group $\mathbb{G} \subseteq E(\mathbb{F}_q)$ is defined by a generator of large prime order such that solving the discrete logarithm problem is hard relative to $\mathbb{G}$.

7.3.4 Class Groups

We focus on class groups of positive definite binary quadratic forms. Let F_Δ denote the set of binary quadratic forms with discriminant Δ. For a given discriminant Δ, we denote the class group by $\mathbb{G}_\Delta$.

An *integral binary quadratic form* is

$$f(x, y) = ax^2 + bxy + cy^2 \tag{7.1}$$

where $a, b, c \in \mathbb{Z}$ are not all equal to zero. We write $f = (a, b, c)$ and call f a *form*. A form is *positive definite* if and only if its discriminant $\Delta = b^2 - 4ac < 0$ and $a > 0$. The group operation is referred to as *composition of forms*. We focus on class groups of positive definite binary quadratic forms and use multiplicative notation.

The class group $\mathbb{G}_\Delta$ is finite and its cardinality is the *class number*, $h(\Delta)$. Computing the class number is at least as hard as factoring the discriminant [BW88]. We will not discuss how to construct class groups for cryptographic purposes, but refer to [CLT18]. For an introduction to form class groups, see for example [BV07, Lon19].

7.3.5 MPC Setting

We consider an MPC setting with m parties tolerating a dishonest majority of up to t passively corrupt parties, $0 \leq t \leq (m-1)/2$. The basic protocols for secure addition and multiplication over a finite field rely on Shamir secret sharing [BGW88]. For our practical experiments we use the MPyC framework [Sch18], which succeeds the VIFF framework [Gei10].

Let $[\![a]\!]$ denote a Shamir secret sharing for any finite field element $a \in \mathbb{F}$. If necessary to specify the field's modulus q, denote $[\![a]\!]_q$ for $a \in \mathbb{F}_q$. The bit length of an integer q is denoted by ℓ_q. In the context of class groups and integer xgcd, $[\![a]\!]$ denotes a Shamir sharing of a bounded integer $a \in \mathbb{Z}$ with $|a| \leq Q$, such that repeated integer multiplication does not flow over the field modulus, q. We will refer to the bit length of q/Q as headroom. A vector is denoted in bold, as well as a vector of shares, e.g., $[\![\mathbf{x}]\!]$. We typically suppress the modulus in the notation of a secret share.

To instantiate secure groups of non-prime order, we also need additive sharings over $\mathbb{Z}$ as well as additive sharings over $\mathbb{Z}_n$ for arbitrary $n > 0$. We use $[\![a]\!]^+_{\mathbb{Z}}$ to denote an additive sharing of $a \in \mathbb{Z}$ and $[\![a]\!]^+_n$ to denote an additive sharing of a modulo n. The shares will be random integers $a_i \in_R \mathbb{Z}_n$ satisfying $\sum_i a_i = a$. We note that a Shamir sharing $[\![a]\!]$ can be converted to an additive sharing $[\![a]\!]^+_p$ by letting each party in a quorum Q, $|Q| \geq t+1$, use $a_i = \lambda_{Q,i} a'_i$ as its additive share, where a'_i is its Shamir share.

Let $\mathcal{P}_i$ for $i \in \{1, ..., m\}$ denote MPC parties. Names of algorithms and protocols are in sans-serif. Let $a \leftarrow \mathsf{open}([\![a]\!])$ denote the protocol for opening a secret share, $[\![r]\!] \leftarrow \mathsf{random}(\mathbb{F})$ the computation of a random Shamir share, $[\![\mathbf{x}]\!] = ([\![a_0]\!], ..., [\![a_{\ell-1}]\!]) \leftarrow \mathsf{bd}([\![a]\!])$ the bit-decomposition of one Shamir share into ℓ_a Shamir shares, $[\![b]\!] \leftarrow \mathsf{ltz}([\![x]\!])$ the less than zero comparison of an (integer) share and $[\![a]\!] \leftarrow \mathsf{norm}([\![\mathbf{x}]\!])$ the determination of the most (or least) significant bit equal to 1. See [Hoo12], [Tof07] and [DFK+06] for these and other common MPC protocols, which we will use as black-boxes.

Secure integer division $([\![q]\!], [\![r]\!]) \leftarrow \mathsf{div}([\![a]\!], [\![b]\!])$, such that $a = bq + r$, is also used as black box. Our implementations are based on the Newton-Raphson method [ACS02, CS10] and Taylor series [DNT12].

7.4 Defining Secure Groups

The secure group scheme includes definitions for representation of secure group elements, secure group operations, including doubling, exponentiation and inversion, random sampling and encoding to/from secure group elements.

Let a map ϱ permit a representation of a group element $a \in \mathbb{G}$ to a tuple or matrix of finite field elements. Note that different finite fields can be used in one representation. Let $[\![a]\!]_{\mathbb{G}}$ denote a secure group element corresponding to coordinate-wise application of $[\![\,]\!]$ to $\varrho(a)$. We will omit ϱ when it is clear from the context that we are using a tuple- or matrix representation, and omit $_{\mathbb{G}}$ from $[\![\,]\!]_{\mathbb{G}}$ and write $[\![a]\!]$ when it is clear that we are using a secure group element.

Definition 20. Let $\mathbb{G}$ be a finite group. A **secure group scheme** comprises protocols for the following tasks, where $a, b \in \mathbb{G}$.

Secure representation. Given $a \in \mathbb{G}$, fields $\mathbb{F}^{(1)}, \ldots, \mathbb{F}^{(j)}$ and map $\varrho : \mathbb{G} \to \mathbb{F}^{(1)} \times \ldots \times \mathbb{F}^{(j)}$, compute $[\![\varrho(a)]\!]_{\mathbb{G}}$.

Secure group operation. Given $[\![a]\!]_{\mathbb{G}}$ and $[\![b]\!]_{\mathbb{G}}$, compute $[\![a * b]\!]_{\mathbb{G}}$.

Secure inversion. Given $[\![a]\!]_{\mathbb{G}}$, compute $[\![a^{-1}]\!]_{\mathbb{G}}$.

Secure exponentiation. Given $[\![a]\!]_{\mathbb{G}}$ and $[\![x]\!]$ with $x \in \mathbb{Z}$, compute $[\![a^x]\!]_{\mathbb{G}}$.

Secure random element. Compute $[\![a]\!]_{\mathbb{G}}$ with $a \in_R \mathbb{G}$.

Secure encoding/decoding. For a set S and an injective map $\sigma : S \to \mathbb{G}$:

- *Encoding.* Given $[\![s]\!]$, compute $[\![\sigma(s)]\!]_{\mathbb{G}}$.
- *Decoding.* Given $[\![a]\!]_{\mathbb{G}}$ with $a \in \sigma(S)$, compute $[\![\sigma^{-1}(a)]\!]$.

Special cases for all these protocols are obtained when one or more inputs/outputs are public. E.g., Section 7.5.5 includes different protocols to compute $[\![b]\!]_{\mathbb{G}} \leftarrow a^{[\![x]\!]}$, $b \leftarrow [\![a]\!]_{\mathbb{G}}^{x}$ and $[\![b]\!]_{\mathbb{G}} \leftarrow [\![a]\!]_{\mathbb{G}}^{[\![x]\!]}$ for $a, b \in \mathbb{G}$ and $x \in \mathbb{Z}$. Furthermore, the protocol to compute $a^{[\![x]\!]}$ with $a \in \mathbb{G}$ of prime order differs from a protocol to compute $a^{[\![x]\!]}$ when $a \in \mathbb{G}$ is of general (known or unknown) order.

Note that there may be multiple encoding/decodings for a group $\mathbb{G}$, each defined on a specific set S. The trivial encoding/decoding is obtained when $S = \mathbb{G}$, which is still interesting in the special cases mentioned above. For example, an encoding (or decoding) with a private input of a group element (held by one of the parties, or an external party) yields a way to perform a private input.

7.4.1 Finite Groups in General

Secure equivalents for the groups $(\mathbb{F}_q, +)$ and $(\mathbb{F}_q^*, *)$ and all their subgroups are available with little implementation effort. It is required that $q > m$, where m is the (maximum) number of parties in the MPC protocol. However, encoding and secure decoding in the threshold setting is not trivial. This is covered in Section 7.5.1.

To extend to other groups, we continue with linear representations, which offer a general technique that permits expressing the group element and operation in an MPC-friendly way.

General Linear Representations of Finite Groups.

Using *linear representations of finite groups* over finite fields, i.e., using the homomorphism $\rho : \mathbb{G} \to GL(V)$ for a vector space V, provides a general way to define secure equivalents for finite groups. For example, for the symmetric group S_q, one could take the permutation matrix representation over $\mathbb{F}_2$, which can be directly translated to its secure equivalent by coordinate-wise application of $[\![\ ]\!]$.

For representation theory and specifically the sub-topic of representations over finite fields, *modular representation theory*, we refer to [Ser77]. General existence of linear representations can be inferred from the fact that every group $\mathbb{G}$ is isomorphic to a subgroup of the symmetric group acting on $\mathbb{G}$, per Cayley's theorem, and that every permutation can be trivially represented by a permutation matrix over a finite field.

For efficiency, we seek linear representations of low degree over small finite fields. This is illustrated with the Rubik's Cube Group in Example 1.[2]

Example 1 (Linear representation of minimum degree)**.** Assume we would like to replace the trusted shuffler in a Rubik's Cube competition by a secure shuffling MPC protocol. The Rubik's Cube group $\mathbb{G}$ is a subgroup of S_{48} generated by the six permutations corresponding to a clockwise turn of one side. For the oblivious linear representation of group elements, we seek a *low* degree linear representation over a *small* finite field.

A candidate linear representation is $\rho : \mathbb{G} \to GL_{20}(\mathbb{F}_7)$. Each $\rho(a)$ for $a \in \mathbb{G}$ corresponds with a permutation matrix that encodes all 20 movable cubies as positions (12 edge and 8 corner cubies). Each position encodes an edge flip or a corner twist as elements in $\mathbb{F}_7$ of multiplicative order 2 and 3, respectively.

[2]Note that small groups can also be implemented using table lookup. The table for $*$ is a public, square matrix X, say. Trivial encoding uses integers in $\{0, \ldots, q-1\}$. Using secure conversion of integers to unit vectors one can evaluate $[\![a * b]\!]$ by $[\![u]\!]^T X [\![v]\!]$, where u and v are the unit vectors corresponding to a and b. Along these lines one can get quite efficient implementations.

The smallest faithful linear representation of the Rubik's cube group is of degree 20 as shown by Holt and Makholm [HM16]. With the above linear representation, the *secure equivalent of a group element* $a \in \mathbb{G}$ corresponds to coordinate-wise application of $[\![\ ]\!]$ to $\rho(a)$, denoted by $[\![a]\!]_{\mathbb{G}}$. The *secure group operation* corresponds to a matrix product based on secure dot products, which can be done at limited cost in many MPC frameworks. This matrix product and the corresponding secure group operation can be executed in one MPC round.

Unsuitability of Linear Representations for Cryptographic Groups. Finding a homomorphism $\rho : \mathbb{G} \to GL(V)$ does not guarantee that the discrete logarithm and decisional Diffie-Hellman problems for the secure group are at least as hard as those for $\mathbb{G}$. Therefore, *for cryptographic applications, we require ρ to be efficient to apply and invert for all $g \in \mathbb{G}$.* The next section discusses secure cryptographic groups in detail.

7.4.2 Secure Cryptographic Groups

This section discusses secure representations of prime order subgroups of $\mathbb{F}_q^*$, elliptic curve groups and class groups.

Prime Order Subgroups of $\mathbb{F}_q^*$.

The representation of prime order subgroups of $\mathbb{F}_q^*$ is straightforward. For $a \in \mathbb{F}_q^*$, define $[\![a]\!]_{\mathbb{G}} := [\![a]\!]$.

Elliptic Curve Groups.

Let $\mathbb{F}$ be a finite field, $[\![\ ]\!] \colon \mathbb{F} \to \mathbb{F}^m$ be a secret sharing for m parties, $\mathbb{G}$ an elliptic curve group defined over $\mathbb{F}$, then the function $\varrho \colon \mathbb{G} \to (\mathbb{F}^m)^2$, $(x, y) \mapsto ([\![x]\!], [\![y]\!])$ trivially maps the affine point $(x, y) \in \mathbb{G}$ to secret shares.

Defining the secure group operation for elliptic curve groups requires expressing the group law formulas without exceptions for any two points. Fortunately, group law formulas without exceptions, so-called *complete* formulas, exist for groups of prime order over Weierstrass curves and Edwards curves. For complete formulas, we refer to [RCB16] for Weierstrass curves and to [BL07] and [HWCD08] for Edwards curves.

Example 2. For an Edwards curve group $\mathbb{G} = E(\mathbb{F}_q)$, we proceed as follows: For a point $P = (x_1, y_1) \in \mathbb{G}$ we define $[\![P]\!]_{\mathbb{G}} := ([\![x_1]\!], [\![y_1]\!])$. The unified group law formula on P and $Q = (x_2, y_2)$ is defined as follows:

$$\left(\frac{x_1y_2 + y_1x_2}{c(1 + dx_1y_1x_2y_2)}, \frac{y_1y_2 - ax_1x_2}{c(1 - dx_1y_1x_2y_2)}\right) = (x_3, y_3). \tag{7.2}$$

Since x_i and y_i ($i \in \{1, 2, 3\}$) are elements of the finite field $\mathbb{F}_q$, operations are defined for Shamir shares. Substituting x_i and y_i by $[\![x_i]\!]$ and $[\![y_i]\!]$ in Eq. (7.2) gives our definition $[\![P]\!]_{\mathbb{G}} + [\![Q]\!]_{\mathbb{G}} \to ([\![x_3]\!], [\![y_3]\!]) = [\![P+Q]\!]_{\mathbb{G}}$.

Efficient Group Law for Secure Edwards Curve Group.

Using twisted Edwards curves, the result from Hisil et al. [HWCD08, Section 4.2] reduces multiplicative depth of our secure group law circuit to two multiplications when parallelized over four processors. Note that the actual number of multiplications is eight plus two scalar multiplications.

The efficient group law requires extended twisted Edwards coordinates. Table 7.1 presents the parallelized formula where (x_1, y_1, t_1, z_1) and (x_2, y_2, t_2, z_2) denote the two inputs in extended coordinates.

Our implementation parallelizes the group law by opening four sets of channels between MPC parties to compute the eight multiplications asynchronously. This also demonstrates a feature of the MPyC framework [Sch18], which natively supports asynchronous functions.

Table 7.1: Parallelized group law formula

Cost	Step	Processor 1	Processor 2	Processor 3	Processor 4
	1	$r_1 \leftarrow y_1 - x_1$	$r_2 \leftarrow y_2 - x_2$	$r_3 \leftarrow y_1 + x_1$	$r_4 \leftarrow y_2 + x_2$
$1M$	2	$r_5 \leftarrow r_1 \cdot r_2$	$r_6 \leftarrow r_3 \cdot r_4$	$r_7 \leftarrow t_1 \cdot t_2$	$r_8 \leftarrow z_1 \cdot z_2$
$1D$	3	$Idle$	$Idle$	$r_7 \leftarrow k \cdot r_7$	$r_8 \leftarrow 2r_8$
	4	$r_1 \leftarrow r_6 - r_5$	$r_2 \leftarrow r_8 - r_7$	$r_3 \leftarrow r_8 + r_7$	$r_4 \leftarrow r_6 + r_5$
$1M$	5	$x_3 \leftarrow r_1 \cdot r_2$	$y_3 \leftarrow r_3 \cdot r_4$	$t_3 \leftarrow r_1 \cdot r_4$	$z_3 \leftarrow r_2 \cdot r_3$

Algorithm 1 [Lon19, Section 6]: compose(f_1, f_2) — f_1, f_2 positive definite

Input: $f_1 = (a_1, b_1, c_1)$ and $f_2 = (a_2, b_2, c_2)$

1: $g \leftarrow 1/2(b_1 + b_2)$, $h \leftarrow -1/2(b_1 - b_2)$
2: $w \leftarrow \mathsf{gcd}(a_1, a_2, g)$
3: $j \leftarrow w$, $s \leftarrow a_1/w$, $t \leftarrow a_2/w$, $u \leftarrow g/w$
4: $(\mu, \nu) \leftarrow$ linear congruence$(tu, hu + sc_1, st)$ ▷ requires computation of the xgcd
5: $(\lambda, \sigma) \leftarrow$ linear congruence$(t\nu, h + t\mu, s)$
6: $k \leftarrow \mu + \nu\lambda, l = (kt - h)/s, m \leftarrow (tuk - hu - c_1 s)/st$
7: **return** $(st, ju - (kt + ls), km - jm)$ ▷ $f_1 * f_2$

Class Groups.

This section discusses secure class group elements and key considerations for implementing the secure group operation for class groups of imaginary quadratic orders. Let $[\![a]\!]$ denote an integer sharing of $a \in \mathbb{Z}$. For a form $f = (a, b, c) \in \mathbb{G}$, a *secure form class group element* is defined as $[\![f]\!]_{\mathbb{G}} := ([\![a]\!], [\![b]\!], [\![c]\!])$ representing an equivalence class. Similar to elliptic curves, the form class group elements are represented by tuples that permit secret sharing and secure arithmetic. However, for class groups we use secure integers instead of finite field elements.

Algorithm 1 presents *composition of two positive definite forms* $f_1, f_2 \in \mathbb{G}_\Delta$. Implementing Algorithm 1 in MPC requires a secure integer gcd protocol in step 2 and a secure integer linear congruence protocol in steps 4 and 5, which in turn requires computation of the extended gcd for integers. The publication of this xgcd protocol is deferred to PRIViLEDGE deliverable D2.4.

Squaring, or composing a form with itself, and repeated squaring are common building blocks for public key cryptography. The composition algorithm simplifies in case of squaring, particularly when the discriminant is the negative of a prime. See Algorithm 2 from [Lon19, Section 6.3.1] for the simplification.

To allow representation of an equivalence class by a unique form and to bound the bit-length of the form, a reduction operation is required. A form $f = (a, b, c)$ is called *normal* if $-a < b \leq a$, and *reduced* if it is normal and $a \leq c$, and if $a = c$ then $b \geq 0$.

The bit-length of the first coefficient doubles after one squaring operation. To reduce the form $f = (a, b, c)$ a *normalization* and a series of *reduction* operations are applied until the criteria for a reduced form are met. In the oblivious case, the number of reduction operations is not allowed to leak information about f, which requires a number of reduction operations that is equal to a general upper bound. We refer to [Lon19, Section 5] for the algorithms. Lemma 6 provides an upper bound for the number of reduction steps after a squaring operation.

Lemma 6 ([Lon19]). *Let $\Delta < 0$ and $f = (a, b, c) \in \mathbb{G}_\Delta$ be a normalized and reduced form. Let $\tilde{f} = (\tilde{a}, \tilde{b}, \tilde{c})$ be the unreduced output of* square*(f) by Algorithm 2. Then the number of reduction steps for $(\tilde{a}, \tilde{b}, \tilde{c})$ is bounded by $\log_2(\sqrt{|\Delta|}/3) + 2$.*

Proof. If $f = (a, b, c)$ is a normalized form, then a maximum of $\log_2(\tilde{a}/\sqrt{|\Delta|}) + 2$ steps is required to produce a reduced form. If $\Delta < 0$, then $a \leq \sqrt{|\Delta|/3}$. From the previous two statements follows that the number of

Algorithm 2 [Lon19]: square(f) — $f \in \mathbb{G}_\Delta$ positive definite, Δ negative prime

Input: $f = (a, b, c)$
1: $(\mu, \nu) \leftarrow$ linear congruence(b, c, a) ▷ requires computation of the xgcd
2: $\tilde{a} \leftarrow a^2, \tilde{b} \leftarrow b - 2a\mu, \tilde{c} \leftarrow \mu^2 - (b\mu - c)/a$
3: **return** $(\tilde{a}, \tilde{b}, \tilde{c})$ ▷ f^2

reduction operations required after one squaring operation is bounded by $\log_2(a^2/\sqrt{|\Delta|})+2 \leq \log_2(\sqrt{|\Delta|}/3)+2$. □

One secure reduction operation requires one secure integer division. From the size of the discriminant then follows the number of secure integer divisions required to keep the form reduced, which becomes large for cryptographic applications. Nevertheless, working with unreduced forms is costlier as it increases the number of rounds for the secure xgcd protocol to quadratic in the size of the unreduced form.

Example 3. Suppose a 1208-bit discriminant provides the security of 2048-bit RSA key, Lemma 6 indicates that an oblivious reduction protocol would require 605 reduction rounds to achieve 2048-bit security.

7.5 Secure Group Protocols

7.5.1 Encoding and Decoding

To enable integer inputs and outputs for applications of secure groups, we consider encoding functions $\sigma\colon S \to \mathbb{G}$ with $S \subset \mathbb{Z}$. We discuss several techniques for encoding an integer s to a group element $\sigma(s)$ and decoding back to s. Depending on the application, either s is secret-shared, or $\sigma(s)$ is secret-shared, or both.

We first present several encodings for quadratic residue groups. Next we consider a generic method for arbitrary groups, and finally, we present a specific encoding for class groups.

Encodings for Quadratic Residues.

Let $\mathbb{G} = (\mathbb{Z}_p^*)^2$ be the group of quadratic residues modulo an odd prime p, and let $n = |\mathbb{G}| = (p-1)/2$. Many encodings (or, embeddings) for $\mathbb{G}$ have been proposed in the cryptographic literature. For our purposes, we consider the following four encodings for $\mathbb{G}$:

$$\begin{aligned}
&\sigma_1\colon \{1,\dots,n\} \to \mathbb{G}, && s \mapsto s^2 \bmod p\\
&\sigma_2\colon \{1,\dots,n\} \to \mathbb{G}, && s \mapsto \begin{cases} s, & \text{if } s \in \mathbb{G}\\ p-s, & \text{if } s \notin \mathbb{G}\end{cases}\\
&\sigma_3\colon \{0,\dots,\lfloor p/k \rfloor - 1\} \to \mathbb{G}, && s \mapsto \min(\mathbb{G} \cap \{ks+i\colon 0 \leq i < k\})\\
&\sigma_4\colon \{1,\dots,n\} \to \mathbb{G}, && s \mapsto H(s, \arg\min_i\{i \geq 1\colon H(s,i) \in \mathbb{G}\}),
\end{aligned}$$

where $1 < k < p$ and H is a cryptographic hash function with codomain $\mathbb{Z}_p^*$.

Encoding σ_1 is the natural encoding for $\mathbb{G}$. Since squaring is a 2-to-1 mapping on $\mathbb{Z}_p^*$ as $s^2 = (-s)^2$, it follows that σ_1 is a bijective encoding on $\{1,\dots,n\}$. Decoding of $a \in \mathbb{G}$ amounts to taking the unique modular square root of $a \leq n$.

For $p \equiv 3 \pmod 4$, σ_2 is another bijective encoding for $\mathbb{G}$, generalizing the encoding defined for safe primes p in [CS03, Section 4.2, Example 2]. We see that $\sigma_2(s) = p - s$ for $s \notin \mathbb{G}$ is indeed a quadratic residue modulo p because $p - s \equiv (-1)s \pmod p$ is the product of two quadratic nonresidues. Decoding of $a \in \mathbb{G}$ amounts to $\sigma_2^{-1}(a) = a$ if $a < p/2$ and $\sigma_2^{-1}(a) = p - a$ otherwise.

Encoding σ_3 resembles an encoding introduced by Koblitz in the context of elliptic curve cryptosystems [Kob87, Section 3.2]. The value of $\sigma_3(s)$ is well-defined as long as each interval $\{ks + i\colon 0 \leq i < k\}$ contains a quadratic residue. This can be ensured by picking k sufficiently large as a function of p. The classical result by

Algorithm 3 $\mathsf{encode}_{\varrho,k}(a)$ $\quad a \in \{0, ..., \lfloor p/k \rfloor\}$

Parameters: Parameter k (see Section 7.5.1), function ϱ as per Definition 20.

1: **repeat** ▷ Loop while no solution $\boldsymbol{a'}$ found in step 4
2: $\quad i \leftarrow_R [0, k)$ ▷ Alternative: $i \leftarrow i + 1$ starting with $i = 0$
3: $\quad a' \leftarrow a \cdot k + i$
4: **until** $\exists\, \boldsymbol{a'}$ such that $\varrho^{-1}(\boldsymbol{a'}) \in \mathbb{G}$ and $\boldsymbol{a'_0} = a'$
5: **return** $\varrho^{-1}(\boldsymbol{a'}), i$ ▷ Output: Encoding in $\mathbb{G}$, increment i (optional)

Burgess [Bur63] implies that $k = \lceil \sqrt{p} \rceil$ ensures successful encoding for sufficiently large p (see also [Hum03]). Under the extended Riemann Hypothesis, Ankeny [Ank52] proved that the least quadratic non-residue for prime p equals $O(\log^2 p)$.

In practice, however, we may set parameter $k = \lceil \log_2 p \rceil$ or a small multiple thereof.[3] In general, encoding σ_3 is not bijective. Decoding of any a in the range of σ_3 is simple as $\sigma_3^{-1}(a) = \lfloor a/k \rfloor$.

Finally, encoding σ_4 corresponds to a hash function used in certain elliptic curve signature schemes [BLS04, Section 3.2]. Since $\Pr[H(s, i) \in \mathbb{G}] = \frac{1}{2}$ in the random oracle model, computing $\sigma_4(s)$ requires two hashes and two Legendre symbols on average. The collision-resistance of H implies that the encoding will be "computationally injective" in the sense that it is infeasible to find $s \neq s'$ for which $\sigma_4(s) = \sigma_4(s')$. Due the one-wayness of H, however, decoding of any a in the range σ_4 amounts to an exhaustive search.

Secure Encoding/Decoding. For our purposes, we are interested in secure computation of these encodings and decodings. We briefly compare the performance for the four encodings. A secure encoding $[\![\sigma_1(s)]\!]$ amounts to a secure squaring of $[\![s]\!]$, which is very efficient. Secure decoding $[\![\sigma_1^{-1}(a)]\!]$ requires taking the modular square root of a quadratic residue $[\![a]\!]$. This can be done efficiently by multiplying $[\![a]\!]$ with a uniformly random square $[\![r]\!]^2$, opening the result ar^2, taking a square root $a^{1/2}r$ (in the clear) and dividing this by $[\![r]\!]$. Finally, we need one secure comparison to make sure that the result is in $\{1, \ldots, n\}$.

Similarly, a secure encoding $[\![\sigma_2(s)]\!]$ amounts to securely evaluating the Legendre symbol $[\![(s \mid p)]\!]$. Since s is known to be nonzero, we simply multiply $[\![s]\!]$ with a uniformly random square $[\![r]\!]^2$ and also with $[\![1 - 2b]\!] = [\![(-1)^b]\!]$ for a uniformly random bit b, opening the result $sr^2(-1)^b$, for which we compute the Legendre symbol $z = (sr^2(-1)^b \mid p)$. Then $[\![(s \mid p)]\!] = z[\![(-1)^b]\!]$. Secure decoding boils down to a secure comparison with public $(p - 1)/2$, which is quite efficient.

A secure encoding $[\![\sigma_3(s)]\!]$ amounts to the secure evaluation of k Legendre symbols $[\![(ks+i \mid p)]\!]$ and finding the first $+1$ among these. With k of the order $\log_2 p$, this represents a considerable amount of work. However, secure decoding $[\![\sigma_3^{-1}(a)]\!] = [\![\lfloor a/k \rfloor]\!]$ is much more efficient, especially if k is a power of two.

Finally, σ_4 is not practical to compute obliviously. To obliviously select the smallest i such that $[\![(H(s, i) \mid p)]\!] = +1$, requires computing $[\![H(s, i)]\!]$ for $0 \leq i < |\mathbb{G}|$, which is not practical. σ_4 does not allow efficient (secure) decoding.

Generic Encodings

Algorithms 3 and 4 generalize encoding σ_3 from Section 7.5.1. Algorithm 3 returns an encoding together with an auxiliary increment i that corresponds to a successful encoding. This simplifies the decoding of a secure group element by using Protocol 1.

[3] Probabilistic heuristics for small primes (< 20 bits), suggest that the greatest number of consecutive quadratic non-residues in the average-case is fit by $\log_2(p)$. Buell and Hudson [BH84] computed the lengths of the longest sequences of consecutive residues and non-residues. By numerical analysis the authors find that the longest sequence of residues and non-residues for a given prime p are fit very closely by $\log_2(p)+\delta$ with δ slightly larger than 1, which suggests a choice of k for an average-case selection of modulus p. A small data set from [Hum03] using smaller primes (`http://www.math.caltech.edu/people/hummel.html`) shows sequences of length $< 2(\log(p) + \delta)$ in worst-case. Heuristics for larger primes (say ≥ 256-bit) are computationally heavy and beyond the scope of this work.

Algorithm 4 $\mathsf{decode}_{\varrho,k}(a, \mathbb{G})$ Assumes encoding with Algorithm 3

Input: $a \in \mathbb{G}$, function ϱ, and parameter k as per Algorithm 3

1: $\boldsymbol{a'} \leftarrow \varrho(a)$
2: $b' \leftarrow \boldsymbol{a'_0}$
3: $b \leftarrow \lfloor b'/k \rfloor$ using integer division
4: **return** b ▷ Output: Decoding b of a with $b \in \{0, ..., \lfloor p/k \rfloor\}$

Protocol 1 $\mathsf{decode}_{\varrho,k}([\![a]\!]_{\mathbb{G}}, [\![i]\!])$ Assumes encoding with Algorithm 3

Input: $[\![a]\!]_{\mathbb{G}}$ for $a \in \mathbb{G}$, representation ϱ, parameter k a power of 2, and (optional) increment $[\![i]\!]$

1: $[\![\boldsymbol{a'}]\!] \leftarrow \varrho([\![a]\!]_{\mathbb{G}})$
2: **if** [[i]] given **then**
3: $\quad [\![b']\!] \leftarrow [\![\boldsymbol{a'_0}]\!] - [\![i]\!]$ ▷ $\boldsymbol{a'_0}$ is the first coefficient of $\boldsymbol{a'}$
4: $\quad [\![b]\!] \leftarrow [\![b']\!]/k$ ▷ $k \mid b'$ holds
5: **else**
6: $\quad [\![b]\!] \leftarrow \mathsf{trunc}([\![\boldsymbol{a'_0}]\!], \log_2(k))$ ▷ truncate (remove) last $\log_2(k)$ bits
7: **return** $[\![b]\!]$ ▷ Output: Decoding of a

This technique is illustrated with an example.

Example 4. Suppose a (twisted) Edwards curve $E(\mathbb{F}_p)$ with curve equation $E : ax^2 + y^2 = c^2(1 + dx^2y^2)$ for given $a, c, d \in \mathbb{F}_p$. For encoding input b, increment $i = 0$ and parameter k, Algorithm 3 step 2 sets $x \leftarrow bk + i$ and tests in step 2 if this x corresponds to an x-coordinate of a valid element $(x, y) \in E(\mathbb{F}_p)$. If not, increase i and repeat. In the case of a (twisted) Edwards curve group, step 2 corresponds to testing if y^2 is a quadratic residue modulo p:

$$(y^2 \mid p) = (\frac{c^2 - ax^2}{1 - c^2dx^2} \mid p) \stackrel{?}{=} +1. \tag{7.3}$$

Protocol 1 decodes a secure group element encoded with Algorithm 3. If the encoder provided a shared auxiliary increment $[\![i]\!]$, secure decoding only requires addition and scalar multiplication. Else, secure decoding requires secure truncation of the least $\log_2(k)$ bits, assuming k is a power of 2.

For class groups, we suggest $k = gap_\ell$, the worst-case prime gap associated with an ℓ-bit discriminant. We explain encoding to class groups in Section 7.5.1.

Encoding to Class Groups.

For class groups of positive definite binary quadratic forms, integer a can be mapped onto form $(a, b, c) \in F_\Delta$ by choosing b as the square root of Δ modulo $4a$. As computation of the square root is more expensive when a is not a prime number, we introduce an encoding approach referred to as *distance embedding* [Sch03]. First, a prime number a' closest to a is selected, then its corresponding square root b' is computed and a' and b' are returned together with distance $a - a'$. Given a class group $\mathbb{G}_\Delta$ the encoding domain has upper bound P, where P is the largest prime P such that $P \leq \sqrt{|\Delta|}/2$.

Holding on to the distance during MPC computations is undesirable if we expect to transform the input a. Given a worst-case prime gap for an ℓ-bit discriminant, gap_ℓ, we avoid the need to return the distance by setting parameter $k = gap_\ell$ and mapping input a to $a' = a \cdot k + i$ for $i = 0$ before starting Algorithm 5. Algorithm 5, step 3 is similar to searching for candidates by incrementing i as in step 2 of Algorithm 3. After a successful encoding of a' per Algorithm 5 to a form $f_{a'} = (a', b, c)$, where a' denotes the prime found in step 3 of Algorithm 5, we can discard the distance knowing that $a = \lfloor a'/gap_\ell \rfloor$.

Algorithm 5 [Sch03]: encode$(n, \mathbb{G}_\Delta)$ — $n \leq P$ for largest prime $P \leq \sqrt{|\Delta|}/2$

$a \leftarrow \max(2, n-1)$
repeat
  $a \leftarrow$ next_prime(a)
until $(\Delta \mid a) = +1 \wedge a \not\equiv 1 \bmod 8$
$b \leftarrow \sqrt{\Delta} \bmod a$ ▷ modular square root
if $\Delta \not\equiv b \bmod 2$ **then**
  $b \leftarrow a - b$
$f \leftarrow (a, b, \frac{b^2-\Delta}{4a})$
$d \leftarrow n - a$
return f, d ▷ Output: Encoding $f \in \mathbb{G}_\Delta$ corresponding to n with distance d

7.5.2 Generating Random Elements in Secure Groups

Dixon [Dix08] presents an algorithm to produce ε-uniformly distributed random elements, meaning that each group element has probability $(1 \pm \varepsilon)(1/|\mathbb{G}|)$ to be the output of the algorithm, for general finite groups.[4] It uses the following definition of random cubes.

Definition 21 (Random Cube, [Dix08])**.** If $a_0, a_1, \ldots, a_{j-1}$ is a list of elements of $\mathbb{G}$, then the *random cube* $W_j := Cube(a_0, \ldots, a_{j-1})$ is the probability distribution where $W_j(b)$ is proportional to the number of ways in which $b \in \mathbb{G}$ can be written as $a_0^{\epsilon_0} \cdots a_{j-1}^{\epsilon_{j-1}}$ with each $\epsilon_i \in \{0, 1\}$.

Using random cubes to sample secure group elements requires k random bits, where k is the security parameter for the required entropy. A protocol that generates k secret elements in $\mathbb{F} \cap \{0, 1\}$, $[\![\boldsymbol{r}]\!]$, is denoted by $[\![\boldsymbol{r}]\!] \leftarrow$ random bits$(\mathbb{F}, k)$.

Theorem 17 ([Dix08])**.** *Let $\{g_0, \ldots, g_{d-1}\}$ be a generating set of $\mathbb{G}$ and $W_j := Cube(g_{j-1}^{-1}, \ldots, g_0^{-1}, g_0, \ldots, g_{j-1})$ a sequence of cubes, where for $j > d$, g_{k-1} is chosen at random from W_{j-1}. Then for each $\delta > 0$, there is a constant K_δ, independent of d or $\mathbb{G}$, such that with probability at least $1 - \delta$, the distribution W_j is $1/4$-uniform when $j > d + K_\delta \log(|\mathbb{G}|)$.*

The number of group operations to construct the random element generator from Theorem 17 is proportional to $\log^2(|\mathbb{G}|)$ and the average cost to produce successive random elements is proportional to $\log(|\mathbb{G}|)$. Precomputation of cube W_j from Theorem 17 can be outsourced to a trusted party or MPC ceremony.

Constant K_δ in Theorem 17 may still make the implementation impractical. However, [Dix08, Theorem 3(c)] states that we can reduce the cube length if we start from a distribution W that is close to the uniform distribution U in the variational distance:

$$||W - U||_{var} := \frac{1}{2} \sum_{a \in \mathbb{G}} |W(a) - U(a)| = \max_{A \subset \mathbb{G}} |W(A) - U(A)|. \tag{7.4}$$

Theorem 18 ([Dix08])**.** *Let U be the uniform distribution on $\mathbb{G}$ and suppose W is a distribution such that $||W - U||_{var} \leq \varepsilon$ for some ε with $0 \leq \varepsilon < 1$. Let $a_0, \ldots, a_{j-1}$ be elements of $\mathbb{G}$ chosen independently according to distribution W. If $Z_j := Cube(a_0, \ldots, a_{j-1})$, then with probability at least $1 - 2^{-h}$, Z_j is 2^{-k}-uniform when*

$$j \geq \frac{1}{\log_2(2/(1+\varepsilon))}(2\log_2(|\mathbb{G}|) + h + 2k). \tag{7.5}$$

Example 5. For the Rubik's Cube group, $\log_2(|\mathbb{G}|) \approx 65.23$. Assuming we generated a $1/4$-uniform cube W using Theorem 17. To generate, with probability at least $1 - 2^{-10}$, a new $1/4$-uniform cube Z_j requires a minimum length of $j \approx 0.47(2 \cdot 65.23 + 10 + 4) \approx 67.90$ group elements. The multiplicative depth of a circuit to generate random Rubik's Cube elements is then $\lceil \log_2 68 \rceil = 7$.

[4] See Lukács [Luk05] for Abelian groups and [MNP01] specifically for class groups.

7.5.3 Secure if-else

Let $c \in \{0, 1\}$, protocol if-else securely selects $[\![a]\!]_{\mathbb{G}}$ or $[\![b]\!]_{\mathbb{G}}$ based on condition $[\![\mathrm{c}]\!]$:

$$\text{if-else}([\![c]\!], [\![a]\!]_{\mathbb{G}}, [\![b]\!]_{\mathbb{G}}) \to [\![a]\!]_{\mathbb{G}} \text{ if } c = 1 \text{ and } [\![b]\!]_{\mathbb{G}} \text{ if } c = 0\ . \tag{7.6}$$

A representation ϱ that maps any element $a \in \mathbb{G}$ to a vector $(a_0, ..., a_{j-1})$ of finite field $\mathbb{F}$ elements, directly permits such a protocol by computing, for each $i \in \{0, ..., j-1\}$ in parallel, $[\![d_i]\!] \leftarrow [\![c]\!] \cdot ([\![a_i]\!] - [\![b_i]\!]) + [\![b_i]\!]$ over $\mathbb{F}$ and then reconstructing the group element $\varrho^{-1}([\![\boldsymbol{d}]\!]) \to [\![d]\!]_{\mathbb{G}}$.

7.5.4 Secure Inverse

The sampling and if-else protocols from Sections 7.5.2 and 7.5.3 allow us to invert secure group elements for finite groups with known or unknown order. Protocol 5 in Appendix 7.8.1 implements this functionality following the classical technique by [BIB89].

7.5.5 Secure Exponentiation for Finite Groups

This section presents secure exponentiation protocols for finite groups of general, known order:

1. Section 7.5.5 for $[\![a^x]\!]_{\mathbb{G}} \leftarrow a^{[\![x]\!]}$
2. Section 7.5.5 for $[\![a^x]\!]_{\mathbb{G}} \leftarrow [\![a]\!]^x_{\mathbb{G}}$
3. Section 7.5.5 for $[\![a^x]\!]_{\mathbb{G}} \leftarrow [\![a]\!]^{[\![x]\!]}_{\mathbb{G}}$

Abelian Groups. The following sections present protocols for abelian groups of prime order (e.g. QR-group $\mathbb{G}_p \subseteq \mathbb{Z}^*_q$) and groups of general, known order (e.g. the Rubik's Cube group). For public groups of prime order, we refer to a technique by [AAN18] in which the multiplicative depth of the protocols does not depend on the input size. We generalize this technique to finite abelian groups by sampling a secure element in the group using Theorem 18.

Non-abelian Groups. For non-abelian groups of known order (e.g. Rubik's Cube group) we cannot apply the random self-reduction techniques of [DFK+06] and [AAN18], but adapt the constant-round secure matrix multiplication protocol by Bar-Ilan and Beaver [BIB89] to secure groups.

Case 1: $[\![a^x]\!] \leftarrow a^{[\![x]\!]}$.

Secure exponentiation, public output. We recall an approach for secure exponentiation with public base and public output, $a^{[\![x]\!]} \to a^x$: First, each party $\mathcal{P}_i$ locally computes $a^{x_i \lambda_i}$, where x_i denotes $\mathcal{P}_i$'s share of $[\![x]\!]$. Then, all parties broadcast the result and compute $\prod_i a^{x_i \lambda_i} \to a^x$. This protocol is valid for $a \in \mathbb{G}_p$ and $x \in \mathbb{Z}_p$.

By using additive share $[\![x]\!]^+$ in the exponent, the above generalizes to groups of general order n. To convert Shamir share $[\![x]\!]$ to additive share $[\![x]\!]^+$, we define convert$([\![x]\!]_n) \to [\![x]\!]^+_n$ as per Section 7.3.5.

Secure exponentiation, secret output. Protocol 2, secure exponentiation, public base, computes $[\![a^x]\!] \leftarrow a^{[\![x]\!]}$. Its round complexity does not depend on the bit length of the inputs, which makes secure computation with large group elements practical. For groups of prime order and Shamir shares that have a modulus p consistent with the order of the base group, Protocol 2 computes the Lagrange interpolation in the exponent.

This generalizes to groups of arbitrary order by converting to additive shares in step 1. For finite groups of order n, convert the Shamir share $[\![\,]\!]_p$ to an additive share $[\![\,]\!]^+_n$ that is consistent with the order of the base group, meaning that elements are in $\mathbb{Z}_n$. The protocol holds for both abelian and non-abelian groups.

Protocol 2 exponentiation$(a, [\![x]\!])$ $a \in \mathbb{G}$, for $x \in \mathbb{Z}_p$

1: Convert [[x]] into additive shares x_i over $\mathbb{Z}_p$, $\mathbb{Z}_n$, or $\mathbb{Z}$ depending on n.
2: Each $\mathcal{P}_i \in Q$ computes $c_i \leftarrow a^{x_i}$.
3: **if** "public output" **then**
4: Each $\mathcal{P}_i \in Q$ publishes c_i.
5: $a^x \leftarrow \prod_{i \in Q} c_i$
6: **return** a^x
7: **else**
8: Each $\mathcal{P}_i \in Q$ distributes $[\![c_i]\!]_{\mathbb{G}}$
9: $[\![a^x]\!]_{\mathbb{G}} \leftarrow \prod_{i \in Q} [\![c_i]\!]_{\mathbb{G}}$
10: **return** $[\![a^x]\!]_{\mathbb{G}}$

Protocol 3 exponentiation$_k([\![a]\!]_{\mathbb{G}}, x)$ $\mathbb{G}$ abelian

Parameters: Security parameter k and cube $Z_k(\mathbf{g})$ as per Theorem 18
1: $[\![\mathbf{r}]\!] \leftarrow$ random bits$(\mathbb{F}, k)$
2: $[\![\boldsymbol{g}^{\boldsymbol{r}}]\!]_{\mathbb{G}} \leftarrow \prod_i$ if-else$([\![r_i]\!], g_i, 1_{\mathbb{G}})$
3: $[\![a * \boldsymbol{g}^{\boldsymbol{r}}]\!]_{\mathbb{G}} \leftarrow [\![a]\!]_{\mathbb{G}} * [\![\boldsymbol{g}^{\boldsymbol{r}}]\!]_{\mathbb{G}}$
4: $a\boldsymbol{g}^{\boldsymbol{r}} \leftarrow$ open$([\![a * \boldsymbol{g}^{\boldsymbol{r}}]\!]_{\mathbb{G}})$
5: $a^x(\boldsymbol{g}^{\boldsymbol{r}})^x \leftarrow (a\boldsymbol{g}^{\boldsymbol{r}})^x$
6: $[\![(\boldsymbol{g}^{\boldsymbol{r}})^{-x}]\!]_{\mathbb{G}} \leftarrow \prod_i$ if-else$([\![r_i]\!], g_i^{-x}, 1_{\mathbb{G}})$
7: $[\![a^x]\!]_{\mathbb{G}} \leftarrow a^x(\boldsymbol{g}^{\boldsymbol{r}})^x * [\![(\boldsymbol{g}^{\boldsymbol{r}})^{-x}]\!]_{\mathbb{G}}$
8: **return** $[\![a^x]\!]_{\mathbb{G}}$

Case 2: $[\![a^x]\!] \leftarrow [\![a]\!]^x$.

Abelian Groups. For a prime-order group $\mathbb{G}_p$, Protocol 7 from [AAN18, Section 3.2] implements secure exponentiation with public exponent efficiently. It calls a sub-protocol for secure exponentiation with public base. Protocol 3 extends this result to finite groups of general order.

In the case of groups of general order, the statistical security of Protocol 3 requires sampling an element ε-uniformly from the group. We use the sampling technique from Section 7.5.2 for general finite groups. Adapting this technique to sample secret elements requires k secret random bits, where k is the security parameter for the required entropy. A protocol that generates k secret elements in $\mathbb{F} \cap \{0, 1\}$, $[\![\boldsymbol{r}]\!]$, is denoted by $[\![\boldsymbol{r}]\!] \leftarrow$ random bits$(\mathbb{F}, k)$.

Non-abelian Groups. The secure exponentiation protocols by [DFK+06, Section 7.2] and [AAN18] do not directly generalize to non-abelian groups. For a random element $r = \prod_i r_i$ and public exponent a, the random self-reduction step in those protocols requires the equality $(\prod_i r_i)^a = \prod_i r_i^a$ to hold.

The constant-round secure matrix multiplication protocol of [BIB89, Section 4.2] does permit a secure exponentiation protocol for non-abelian groups, given that we can sample ε-uniformly random and compute the inverse for secure groups (see Sections 7.5.2 and 7.5.4 respectively). Protocol 6 in Appendix 7.8.1 presents this direct generalization of [BIB89, Section 4.2].

Case 3: $[\![a^x]\!] \leftarrow [\![a]\!]^{[\![x]\!]}$.

Abelian Groups. Analogous to the previous protocol, we generalize Protocol 8 from [AAN18, Section 3.3] to abelian groups of known order. Protocol 3 is changed slightly to enable secure exponentiation with secret base and secret exponent. Replace step 5 by $[\![a^x(\boldsymbol{g}^{\boldsymbol{r}})^x]\!]_{\mathbb{G}} \leftarrow (a\boldsymbol{g}^{\boldsymbol{r}})^{[\![x]\!]}$ using Protocol 2. Replace step 6 by $[\![(\boldsymbol{g}^{\boldsymbol{r}})^{-x}]\!]_{\mathbb{G}} \leftarrow \prod_i$ if-else$([\![r_i]\!], g_i^{[\![-x]\!]}, 1_{\mathbb{G}})$, using k parallel instances of Protocol 2 to compute $g_i^{[\![-x]\!]}$. The protocol then returns the multiplication of the outcomes of Steps 5 and 6.

Non-abelian Groups. Protocol 7 in Appendix 7.8.1 generalizes the bit-decomposition technique of [DFK+06, Section 7.2] to non-abelian groups. For secret base $[\![a]\!]_{\mathbb{G}}$ and secret exponent $[\![x]\!]$ of bit length ℓ_x, it calls Protocol 6 ℓ_x times (in parallel) to compute $[\![a]\!]_{\mathbb{G}}^{2^j}$ for $j \in \{0, \ldots, \ell_x\}$.

7.5.6 Secure Exponentiation for Groups of Unknown Order

Protocol 7 applies to groups of unknown order, e.g. class groups. For secure exponentiation with secret base, we call Protocol 6, which requires sampling from the group (e.g. using Theorem 18).

7.5.7 Privacy and Correctness of the Secure Group Protocols

Privacy. Statistical security follows from the fact that Protocol 3 is an appropriate secure random self-reduction of Protocol 2, if protocol secure random samples uniformly random from the group. Using Theorem 18, sampling is ϵ-uniform for arbitrarily small ϵ. Hence, Protocol 3 can be considered a functionality of an Arithmetic Black Box. Under the same caveat of ϵ-uniform, Protocols 5, 6 and 7 can be considered applications of an Arithmetic Black Box.

Correctness. Our protocols are direct generalizations of techniques presented in [AAN18], [DFK+06] and [BIB89].

7.6 MPC-blended Threshold Cryptosystems

In this section we present several applications of secure groups in the context of threshold cryptosystems, using the well-known threshold ElGamal cryptosystem as a basic example [Ped91a]. We focus on the case of security against passive adversaries, assuming an MPC setting with m parties and a corruption threshold of t, $0 \leq t < m/2$.

Let $\mathbb{G}$ be a cyclic group with generator g of large prime order p. Given our protocols for secure groups, a simple $(t+1, m)$-threshold ElGamal cryptosystem is obtained as follows:

Distributed key generation. The parties generate $[\![x]\!]$ with $x \in_R \mathbb{Z}_p$, and run Protocol 2 to compute g^x. The parties keep private key $[\![x]\!]$ in shares and output public key $h = g^x$.

Encryption. Given message $M \in \mathbb{G}$, pick $u \in_R \mathbb{Z}_n$. The ciphertext for public key h is the pair $(g^u, h^u M)$.

Threshold decryption. Given ciphertext (A, B), the parties use $[\![x]\!]$ to run Protocol 2 to compute A^x. The parties output message $M = B/A^x$.

The complexity of the protocols for distributed key generation and threshold decryption is entirely hidden in the underlying MPC framework supporting secure groups.

Note: for Protocol 2 only a quorum of $t+1$ parties are needed, each publishing a^{x_i} (in MPyC, use mpc.transfer() from $t+1$ parties to all m parties).

We now extend the threshold ElGamal cryptosystem noting that instead of using message $M \in \mathbb{G}$ in the clear we can also use $[\![M]\!]_{\mathbb{G}}$ in shares, just as we keep the private key $[\![x]\!]$ in shares, using our protocols for secure groups:

Encryption of shared message. Given message $[\![M]\!]_{\mathbb{G}}$, the parties generate $[\![u]\!]$ with $u \in_R \mathbb{Z}_p$, and output ciphertext for public key h as the pair $(g^{[\![u]\!]}, h^{[\![u]\!]}[\![M]\!]_{\mathbb{G}})$. Here, Protocol 2 is used twice, either with public output (A, B) or with secret output $([\![A]\!]_{\mathbb{G}}, [\![B]\!]_{\mathbb{G}})$.

Threshold decryption to shared message. Given ciphertext (A, B), the parties run Protocol 2 to compute $[\![A^x]\!]_{\mathbb{G}} = A^{[\![x]\!]}$. The parties compute and keep message $[\![M]\!]_{\mathbb{G}} = B/[\![A^x]\!]_{\mathbb{G}}$ in shares. Similarly, the parties may decrypt a ciphertext $([\![A]\!]_{\mathbb{G}}, [\![B]\!]_{\mathbb{G}})$.

Protocol 4 convert$(E_h(a))$ — $E_h(a)$ an ElGamal encryption, $h = g^x$

Input: $E_h(a) = (c_1, c_2) \in \mathbb{G} \times \mathbb{G}$, secret shared $[\![x]\!]$ such that $h = g^x$

1: $[\![c_1^x]\!]_{\mathbb{G}} \leftarrow c_1^{[\![-x]\!]}$ with Protocol 2
2: $[\![a]\!]_{\mathbb{G}} \leftarrow c_2 [\![c_1^{-x}]\!]_{\mathbb{G}}$
3: $[\![b]\!] \leftarrow$ decode$([\![a]\!]_{\mathbb{G}})$ ▷ Using Protocol 1
4: **return** $[\![b]\!]$ ▷ Output: b such that $\sigma(b) = a$

Combining these protocols we can do things like reencryption for another public key h': use threshold decryption to a shared message and then encryption under the new public key. We can also do a reencryption as follows:

Proxy reencryption. Assume the parties hold shares for two private keys $[\![x_1]\!]$ and $[\![x_2]\!]$, then they may compute $[\![x_1/x_2]\!]$ and use this with Protocol 2 to convert ciphertext (A, B) for public key $h_1 = g^{x_1}$ into ciphertext $(A^{[\![x_1/x_2]\!]}, B)$ for public key $h_2 = g^{x_2}$.

Proxy reencryption key. Assume the parties hold shares for two private keys $[\![x_1]\!]$ and $[\![x_2]\!]$, then they may compute and open $[\![x_1/x_2]\!]$ as a proxy reencryption key.

The above protocols can be extended with encoding/decoding. Protocol 4 convert extends the functionality threshold decryption to shared message by outputting a Shamir share $[\![b]\!]$ instead of a secure group element, assuming the input of the encryption is encoded with Algorithm 3. For readability Protocol 4 assumes no auxiliary input (e.g. increment i) is passed to decode in Step 3.

To pass increment i securely to MPC parties using the extended threshold cryptosystem, we slightly adapt σ_3 from Section 7.5.1. We modify σ_3 as follows: $\sigma_{3'} \colon \{0, \ldots, \lfloor p/k \rfloor - 1\} \to (\mathbb{G}, \mathbb{G}), s \mapsto (ks + i, i)$ such that $i = \arg\min_i \{0 \leq i < k \colon ks + i \in \mathbb{G} \cap i \in \mathbb{G}\}$. Intuitively, the output of this $\sigma_{3'}$ can be viewed as $(\sigma_3(s), \sigma_3(0))$ where i is equal for both (ignoring that i is the smallest increment encoding $\sigma_3(s)$). We now send encryptions of both group elements to MPC parties. To decode $\sigma_3(s)$, parties interpret the first coordinate of $[\![\sigma_3(0)]\!]_{\mathbb{G}}$ as $[\![i]\!]$ and pass it to decode in Step 3 of convert.

7.7 Implementation

The *secure groups scheme* is implemented in Python using the MPyC package [Sch18]. This implementation includes secure representations for various elliptic curve groups (Weierstrass, Edwards), the symmetric group, the group of quadratic residues and class groups. To demonstrate its applicability, we include demos for the threshold conversion protocol (Protocol 4) using quadratic residues and twisted Edwards curves, computations with class groups using a new xgcd protocol (under development), random sampling in the Rubik's cube group, the well-known Pinocchio ZK-SNARK [PHGR13] and the Trinocchio multiparty zero knowledge proof [SVdV16] based on Pinocchio. The Secure Groups Python package will be published with the final version of this work.

7.8 Appendix

7.8.1 Exponentiation Protocols for Non-abelian Groups

For completeness, the secure exponentiation protocols that are a direct translation of the protocols by [BIB89] and [DFK+06] are summarized below.

Protocol 6 has a number of rounds that does not depend on the bit length of the input. However, while step 4 requires a constant number of rounds, the number of invocations is linear in the size of the exponent, ℓ_x. The number of invocations can be reduced to $O(\log(\ell_x))$ by applying repeated squaring. This increases the round complexity to $O(\log(\ell_x))$.

Protocol 5 [BIB89]: $\mathsf{inverse}_k([\![a]\!]_{\mathbb{G}})$ — $\mathbb{G}$ abelian or non-abelian

Parameters: Security parameter k and cube $Z_k(\mathbf{g})$ as per Theorem 18

$[\![\mathbf{r}]\!] \leftarrow \mathsf{random\ bits}(\mathbb{F}, k)$
$[\![\boldsymbol{g^r}]\!]_{\mathbb{G}} \leftarrow \prod_i \mathsf{if\text{-}else}([\![r_i]\!], g_i, 1_{\mathbb{G}})$
$\boldsymbol{g^r} a \leftarrow \mathsf{open}([\![\boldsymbol{g^r}]\!]_{\mathbb{G}}[\![a]\!]_{\mathbb{G}})$
$a^{-1}(\boldsymbol{g^r})^{-1} \leftarrow (\boldsymbol{g^r} a)^{-1}$
$[\![a^{-1}]\!]_{\mathbb{G}} \leftarrow a^{-1}(\boldsymbol{g^r})^{-1}[\![\boldsymbol{g^r}]\!]_{\mathbb{G}}$
return $[\![a^{-1}]\!]_{\mathbb{G}}$

Protocol 6 [BIB89]: $\mathsf{exponentiation}_k([\![a]\!]_{\mathbb{G}}, x)$ — $\mathbb{G}$ abelian or non-abelian

Parameters: Security parameter k and cube $Z_k(\mathbf{g})$ as per Theorem 18

$\{[\![r_{0,0}]\!], [\![r_{0,1}]\!], \ldots, [\![r_{0,k-1}]\!], \ldots, [\![r_{x-1,k-1}]\!]\} \leftarrow \mathsf{random\ bits}(\mathbb{F}, x \cdot k)$
$[\![\boldsymbol{g^{r_j}}]\!]_{\mathbb{G}} \leftarrow \prod_i \mathsf{if\text{-}else}([\![r_{i,j}]\!], g_i, 1_{\mathbb{G}})$ for $j \in \{0, \ldots, x-1\}$
$[\![(\boldsymbol{g^{r_j}})^{-1}]\!]_{\mathbb{G}} \leftarrow \mathsf{inverse}([\![\boldsymbol{g^{r_j}}]\!]_{\mathbb{G}})$ for $j \in \{0, \ldots, x-1\}$ ▷ using Protocol 5
$[\![b_j]\!]_{\mathbb{G}} \leftarrow [\![(\boldsymbol{g^{r_j - 1}})^{-1}]\!]_{\mathbb{G}}[\![a]\!]_{\mathbb{G}}[\![(\boldsymbol{g^{r_j}})]\!]_{\mathbb{G}}$ for $j \in \{0, \ldots, x-1\}$
$b_j \leftarrow \mathsf{open}([\![b_j]\!]_{\mathbb{G}})$ for $j \in \{0, \ldots, x-1\}$
$b \leftarrow \prod_j b_j$
$[\![a^x]\!]_{\mathbb{G}} \leftarrow [\![\boldsymbol{g^{r_0}}]\!]_{\mathbb{G}}\ b\ [\![\boldsymbol{g^{r_x - 1}}]\!]_{\mathbb{G}}$
return $[\![a^x]\!]_{\mathbb{G}}$

Protocol 7 $\mathsf{exponentiation}_k([\![a]\!]_{\mathbb{G}}, [\![x]\!])$ — $\mathbb{G}$ abelian or non-abelian

Parameters: Security parameter k and cube $Z_k(\mathbf{g})$ as per Theorem 18

$\{[\![b_0]\!], \ldots, [\![b_{\ell_x}]\!]\} \leftarrow \mathsf{bd}([\![x]\!])$
$[\![c_j]\!]_{\mathbb{G}} \leftarrow [\![a]\!]_{\mathbb{G}}^{2^j}$ for $j \in \{0, \ldots, \ell_x\}$ ▷ using Protocol 6 with parameter k
$[\![d_j]\!]_{\mathbb{G}} \leftarrow \mathsf{if\text{-}else}([\![b_j]\!], [\![c_j]\!]_{\mathbb{G}}, [\![1]\!]_{\mathbb{G}})$ for $j \in \{0, \ldots, \ell_x\}$
$[\![a^x]\!]_{\mathbb{G}} \leftarrow \prod_{j \in \{0, \ldots, \ell_x\}} [\![d_j]\!]_{\mathbb{G}}$
return $[\![a^x]\!]_{\mathbb{G}}$

Chapter 8

Server-Assisted Hash-Based Signature Schemes

In this chapter, we introduce the *BLT* family of signature schemes (named after the initials of the inventors). The schemes combine hash function based authentication with time-stamping. As the latter can also be built from hash functions (following the hash-then-publish model), it can be argued that hash functions are the sole underlying cryptographic primitive. The chapter is based on the following publications:

[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.

[BFL$^+$19] 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.

[BFLT20] Ahto Buldas, Denis Firsov, Risto Laanoja, and Ahto Truu. Verified security of BLT signature scheme. In *ACM SIGPLAN CPP 2020, Proceedings*, pages 244–257. ACM, 2020.

8.1 Motivation and Related Work

All the digital signature schemes in wide use today (RSA [RSA78], DSA [ElG85b], ECDSA [JMV01]) are known to be vulnerable to quantum attacks by Shor's algorithm [Sho99]. While the best current experimental results are still toy-sized [MLLL$^+$12], 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. These considerations triggered the Post-Quantum Cryptography project, announced by NIST in December 2016 [NIS16].

Error-correcting codes, lattices, and multi-variate polynomials have been used as foundations for proposed replacement schemes [BBD09]. 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 quite resilient to quantum attacks. The best currently 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 its adaptation by Brassard *et al.* [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. The crucial point is that the attack costs remain exponential even for quantum adversaries.

Another advantage of hash function based schemes is the minimization of assumptions. It has been shown that secure digital signatures can exist if and only if one-way and second pre-image resistant hash functions exist [Rom90]. As the signature schemes we study in the following depend on the hash function having additional properties, they are formally not based on minimal assumptions. However, all real-life deployments that we are aware of implement the hash-then-sign model that relies on a collision resistant hash function in addition to a signature scheme. Therefore, it can be argued that in practice, collision resistance should also be counted among the minimal assumptions.

The proposed schemes are really templates that can be instantiated with different hash functions. A benefit of this approach is that when the security of one hash function becomes insufficient (whether by advances in cryptoanalytic techniques or in computing power, or a combination of the two), we can replace the hash function with a stronger one and the security of the new instantiation of the signature scheme follows from the security of the replacement hash function.

8.2 Summary

In Section 8.3 we describe the the foundations of the scheme—cryptographic hash functions and hash-and-publish time-stamping.

In Section 8.4 we describe the scheme we call BLT-TB (for "time-bound") that essentially consists of authenticating messages with one-time keys pre-bound to fixed time slots and proving correct usage of these keys by time-stamping the authenticators. The main drawback of the scheme is the need to pre-generate keys for all possible signing times, which can be prohibitive for constrained devices, such as smart cards.

In Section 8.5 we propose a way to reduce the key generation costs for personal devices that are used only occasionally. The BLT-BC scheme (for "blockchain") makes the keys one-time and uses a blockchain-backed validator (perhaps implemented as a consensus-based cluster) to enforce the one-time property of the signing keys. Thus, the scheme trades savings on the client side against higher requirements on the supporting service, both in the trust placed on the service and also the computing resources required to operate it.

In Section 8.6 we propose another way to relax the requirement to pre-bind the keys to time by developing the concept of forward-resistant tags that combined with a backdating-resistant time-stamping service yields an unforgeable signature scheme. We then propose several forward-resistant tag systems, in particular some that allow dynamic binding of keys to time, and derive their security properties from those of the underlying hash function. Based on that, we then define the BLT-OT (for "one-time") signature scheme where each key can be used just once, but at the time of keyholder's choosing. The resulting scheme has competitive performance both as a one-time scheme and also as a component of a many-time scheme.

All these signature schemes depend on access to a time-stamping service and are thus inherently server-supported. On one hand this comes with the need to critically trust an external service for the security of the signatures and the restriction that signing is only possible on-line. On the other hand, most practical deployments of digital signatures in open systems need to track key revocations and are on-line even when the underlying signature scheme theoretically supports off-line signing and the server-supported nature of our schemes also provides a natural choke point for implementing instant key revocation and possibly enforcing other kinds of usage policies as well.

8.3 Preliminaries

8.3.1 Hash Functions

In general, a hash function h maps arbitrary-sized input data to fixed-size output values:

$$h : \{0,1\}^* \to \{0,1\}^k \ .$$

Even though some actual hash functions technically are defined only for inputs up to a specific length, these limits are so high that for all practical purposes the functions can be considered unlimited.

Hash values are often used as representatives of data that are either too large or too confidential to be used directly. For example, in the hash-then-publish time-stamping model, a hash value of a document is published to establish evidence of the existence of the document without disclosing its contents. Likewise, in the hash-then-sign model, a document's hash value is signed instead of the document itself.

To facilitate such uses, cryptographic hash functions are expected to have several additional properties, such as one-wayness (it's infeasible to reconstruct the input from the output), second pre-image resistance (it's infeasible to change the input so that it still maps to the same output), and collision resistance (it's infeasible to find two distinct inputs mapping to the same output). These properties have received extensive formal treatment in [RS04, ANPS07, AS11].

8.3.2 Hash Trees and Hash Chains

Introduced by Merkle [Mer79], a hash tree is a tree-shaped data structure built using a 2-to-1 hash function $h : \{0,1\}^{2k} \to \{0,1\}^k$. The nodes of the tree contain k-bit values. Each node is either a leaf with no children or an internal node with two children. The value x of an internal node is computed as $\{x\}\, h(x_l, x_r)$, where x_l and x_r are the values of the left and right child, respectively. There is one root node that is not a child of any node. We will use $\{r\}\, T^h(x_1, \ldots, x_N)$ to denote a hash tree whose N leaves contain the values $x_1, \ldots, x_N$ and whose root node contains r.

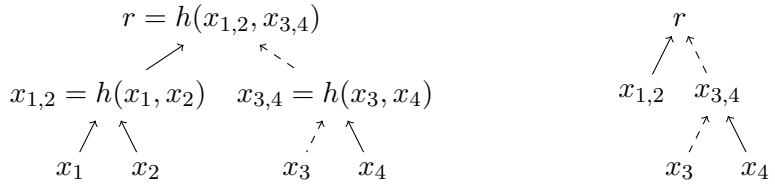

Figure 8.1: The hash tree $T^h(x_1, \ldots, x_4)$ and the corresponding hash chain $x_3 \rightsquigarrow r$.

In order to prove that a value x_i participated in the computation of the root hash r, it is sufficient to present values of all the sibling nodes on the unique path from x_i to the root in the tree. For example, to claim that x_3 belongs to the tree shown on the left in Figure 8.1, one has to present the values x_4 and $x_{1,2}$ to enable the verifier to compute $\{x_{3,4}\}\, h(x_3, x_4)$, $\{r\}\, h(x_{1,2}, x_{3,4})$, essentially re-building a slice of the tree, as shown on the right in Figure 8.1. We will use $x \overset{c}{\rightsquigarrow} r$ to denote that the hash chain c links x to r in such a manner.

Intuitively, it seems obvious that if the function h is collision-resistant, the existence of such a chain whose output equals the original r is a strong indication that x was indeed the original input. However, this result was not formally proven until 25 years after the hash tree construct was proposed [BS04, Cor05].

8.3.3 Hash-Then-Publish Time-Stamping

Following Buldas and Saarepera [BS04], we model the hash-and-publish time-stamping service as consisting of a repository $\mathcal{R}$ and an aggregation layer S (Figure 8.2). We consider the repository $\mathcal{R}$ to be an ideal object that works as follows:

- The time t is initialized to 1, and all the cells $\mathcal{R}_i$ to $\perp$.
- The query $\mathcal{R}.\texttt{time}$ is answered with the current value of t.
- The query $\mathcal{R}.\texttt{get}(t)$ is answered with R_t.
- On the request $\mathcal{R}.\texttt{put}(x)$, first $\mathcal{R}_t \leftarrow x$ is assigned and then $t \leftarrow t + 1$.

The aggregation layer S operates in fixed-duration rounds. During each round, S collects client requests. At the end of the round, S aggregates the received requests $x_1, \ldots, x_N$ into a hash tree $r \leftarrow T^h(x_1, \ldots, x_N)$, queries t via a call to $\mathcal{R}.\texttt{time}$, commits the root r of the hash tree via $\mathcal{R}.\texttt{put}(r)$, and finally returns to each client the hash chain a linking that client's input x to the committed root r.

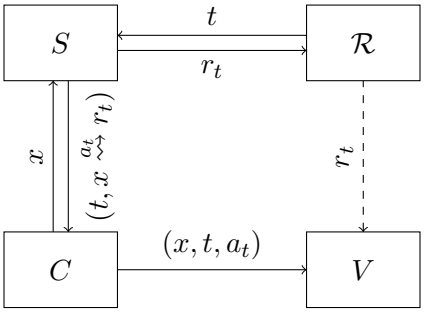

Figure 8.2: Interactions in the hash-then-publish time-stamping between the client C, the aggregation service S, the ideal repository $\mathcal{R}$, and the verifier V.

A verifier V, receiving an input x and a time-stamp (t, a_t), first obtains r_t by querying $\mathcal{R}.\texttt{get}(t)$ and then checks that the hash chain a_t links the input x to r_t.

To simplify presentation, we count time in aggregation rounds of the time-stamping service and use the expression "at time t" to mean "during aggregation round t".

The security of such schemes can be proven in a model where only the repository $\mathcal{R}$ is assumed to operate correctly and the service S does not have to be trusted. We refer to [BS04, BN10, BL13, BLLT14] for detailed analyses, including the requirements on the hash function h used by S in the general setting, and list our assumptions case by case in the proofs in the following sections.

8.4 Time-Stamped Scheme with Time-Bound Keys

The principal idea of our first scheme [BLT17] is to have the signer commit to a sequence of keys so that each key is assigned a time slot when it can be used to sign messages and will transition from a signing key to a verification key once the time slot has passed. Signing itself then consists of time-stamping the message-key pair in order to prove that the signing operation was performed at the correct time.

8.4.1 Description of the Scheme

More formally, the procedures for key generation, signing, and verification are as follows.

Key Generation. To prepare to sign messages at times $1, \ldots, T$, the signer:

1. Generates T unpredictable k-bit values as signing keys: $(z_1, \ldots, z_T) \leftarrow \mathcal{G}(T, 1^k)$.
2. Binds each key to its time slot: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, T\}$.
3. Aggregates the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_T)$.
4. Publishes the root hash p as the public key.

The resulting data structure is shown in Figure 8.3 and its purpose is to be able to extract hash chains c_i such that $h(i, z_i) \overset{c_i}{\rightsquigarrow} p$ for $i \in \{1, \ldots, T\}$.

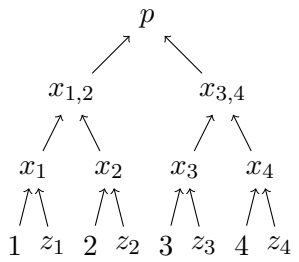

Figure 8.3: Computation of the public key for $T = 4$: $z_1, \ldots, z_4$ are the private keys.

Signing. To sign message m at time t, the signer:

1. Uses the appropriate key to authenticate the message: $y \leftarrow h(m, z_t)$.
2. Time-stamps the authenticator: sends y to the time-stamping service and receives in response a_t such that $y \overset{a_t}{\rightsquigarrow} r_t$, where r_t is the root hash of the aggregation tree built by the time-stamping service for the aggregation round t. We assume the root hash is committed to in some reliable way, such as broadcasting it to all interested parties, but place no other trust in the service.
3. Outputs the tuple (t, z_t, c_t, a_t), where t is the signing time, z_t is the signing key for time slot t, c_t is the hash chain linking the binding (t, z_t) to the signer's public key p, and a_t is the hash chain from the time-stamping service linking the message-key pair (m, z_t) to the round commitment r_t.

Note that the signature is composed and emitted after the time-stamping step, which makes it safe for the signer to release the key z_t as part of the signature: the time-stamping aggregation round t has ended and any future uses of the key z_t can no longer be stamped with time t.

Verification. To verify that the message m and the signature $s = (t, z, c, a)$ match the public key p, the verifier:

1. Checks that z was committed as signing key for time t: $h(t, z) \overset{c}{\rightsquigarrow} p$.
2. Checks that m was authenticated with key z at time t: $h(m, z) \overset{a}{\rightsquigarrow} r_t$.

8.4.2 Implementation Considerations

Key Generation. In the description of the scheme we assumed that the signing keys $z_1, \ldots, z_T$ are unpredictable values drawn from $\{0, 1\}^k$, but left unspecified how they might be generated in practice. Obviously they could be generated as independent truly random values, but this would be rather expensive and also would necessitate keeping a large number of secret values over a long time. It would be more practical to generate them pseudo-randomly from a single random seed s. There are several known ways of doing that:

- Iterated hashing: $z_T \leftarrow s$, $z_{i-1} \leftarrow h(z_i)$ for $i \in \{2, \ldots, T\}$.

  This idea of generating a sequence of one-time keys from a single seed is due to Lamport [Lam81] and has also been used in the TESLA protocol [PCTS02]. Implemented this way, our scheme would also bear some resemblance to the Guy Fawkes protocol [ABC+98].

  Note that the keys have to be generated in reverse order, otherwise the earlier keys released as signature components could be used to derive the later ones that are still valid for signing. To be able to use the keys in the direct order, the signer would have to either remember them all, re-compute half of the sequence on average, or implement a traversal algorithm such as the one proposed by Schoenmakers [Sch17].

- Counter hashing: $z_i \leftarrow h(s, i)$ for $i \in \{1, \ldots, T\}$.

  With a hash function behaving as a random oracle, this scheme would generate keys indistinguishable from truly random values, but we are not aware of any strong results on the security of practical hash functions when used in this mode.

- Counter encryption: $z_i \leftarrow E_s(i)$ for $i \in \{1, \ldots, T\}$.

  The signing keys are generated by encrypting their indices with a symmetric block cipher using the seed as the encryption key. This is equivalent to using the cipher in the counter mode as first proposed by Diffie and Hellman [DH79]. The security of this mode is extensively studied for all practical block ciphers. Another benefit of this approach is that it can be implemented using standard hardware security modules where the seed is kept in protected storage and the encryption operations are performed in a security-hardened environment.

Time-Stamping. As already mentioned, we side-step the key state management problems [MKF+16] common for most N-time signing schemes by making the signing keys not one-time, but *time-bound* instead. This in turn raises the issue of clock synchronization.

We first note that even when the signer's local clock is running fast, premature key release is easy to prevent by having the signer verify the time-stamp on $h(m, z_t)$ before releasing z_t.

The next issue is that the signer needs to select the key z_t before computing $h(m, z_t)$ and submitting it to time-stamping. If, due to clock drift or network latency, the time in the time-stamp received does not match t, the signature can't be composed. To counter clock drift and stable latency, the signer can first time-stamp a dummy value and use the result to compare its local clock to that of the time-stamping service.

To counter network jitter, the signer can compute the message authenticators $h(m, z_{t'})$ for several consecutive values of t', submit all of them in parallel, and compose the signature using the components whose t' matches the time t in the time-stamps received. Buldas *et al.* [BKL13] have shown that with careful scheduling the latency can be made stable enough for this strategy even in an aggregation network with world-wide scale.

Finally, we note that time-stamping services operating in discrete aggregation rounds are particularly well suited for use in our scheme, as they only return time-stamps once the round is closed, thus eliminating the risk that a fast adversary could still manage to acquire a suitable time-stamp after the signer has released a key.

8.4.3 Discussion

Performance. In the following estimates, we assume the use of SHA-256 [NIS01], a common 256-bit hash function. On small inputs, a moderate CPU core can perform about a million SHA-256 evaluations per second.[1] We also assume a signing key sequence containing one key per second for a year, for a total of a bit less than 32 million, or roughly 2^{25} keys.

Using the techniques described above, generation of T signing keys takes T applications of either a hash function or a symmetric block cipher. Aggregating them into a public key takes $2N - 1$ hashing operations. Thus, the key generation in our example takes about 100 seconds on a single core (and is well parallelizable if either of the counter-based generator mechanisms is used).

The resulting public key consists of just one hash value. In the private key, only the seed s has to be kept secret. The signing keys $z_1, \ldots, z_T$ can be erased once the public key has been computed, and then re-generated as needed for signing.

The hash tree $T^h(x_1, \ldots, x_T)$ presents a space-time trade-off. It may be kept (in regular unprotected storage, as it contains no sensitive information), taking up $2N - 1$ nodes, or about 1 GB, and then the key authentication hash chains can be just read from the tree with no additional computations needed. Alternatively, one can use a hash tree traversal algorithm, such as the one proposed by Szydlo [Szy04], to keep only $3 \log_2 N$ nodes of

[1] As reported by OpenSSL 1.0.2 speed test on a laptop with 2.3 GHz Intel Core i5 CPU.

the tree and spend $2\log_2 N$ hash function evaluations per chain extraction, assuming all chains are extracted consecutively.

The size of the signature (t, z_t, c_t, a_t) is dominated by the two hash chains. The key authentication chain consists of $\log_2 N$ hash values, for a total of about 800 B for our 1-year key sequence. The time-stamping chain consists of $\log_2 M$ hash values, where M is the number of requests received by the time-stamping service in the round t. Assuming the use of the KSI service described in [BKL13] operating at its full capacity of 2^{50} requests per round, this adds about 1 600 B. Thus we can expect signatures of less than 3 kB.

As the verification means re-computation of the hash chains, it amounts to less than a hundred hash function evaluations.

We will compare the performance of this scheme to the state of the art in Section 8.6.3.

Security Model. The security model of our scheme is markedly different from that of the "standalone" schemes. The correct operation of the repository $\mathcal{R}$ backing the time-stamping service is critical for the unforgeability of the signatures. An adversary that can tamper with the contents of $\mathcal{R}$ could back-date time-stamps and thus reuse keys already released as components of legitimate signatures to create forgeries.

There are practical ways to mitigate this risk. For example, the repository could be based on a data structure where each record includes a hash value of the preceding one and new records could be added by consensus among multiple independently operated nodes, essentially implementing a distributed robust public transaction ledger [GKL15]. This, however, poses additional engineering challenges when deploying the scheme in practice.

Also the verifiers of signatures need to be able to authenticate the responses received from the repository when they query the round commitments.

8.5 Blockchain-Backed Scheme with One-Time Keys

The signing keys in the scheme proposed in Section 8.4 are really not one-time, but rather time-bound: every key can be used for signing only within a specific time interval. For that reason, we will refer to the scheme as BLT-TB (for "time-bound") in the following. The architecture of the BLT-TB scheme can be modeled as interactions between the following parties (Figure 8.4, left):

- The signer who uses trusted functionality in secure device D to manage private keys.
- Server S that aggregates key usage events from multiple signers in fixed-length rounds and posts the summaries to append-only repository $\mathcal{R}$.
- Verifier V who can verify signatures against the signer's public key p and the round summaries r_t obtained from the repository.

Note that S and $\mathcal{R}$ together implement a hash-then-publish time-stamping service where neither the signer nor the verifier needs to trust S; only $\mathcal{R}$ has to operate correctly for the scheme to be secure.

8.5.1 Design of the Scheme

One-Time Keys. The design of BLT-TB incurs quite a large overhead as keys must be pre-generated even for time periods when no signatures are created. To avoid this inherent inefficiency, we now propose [BLT18] to spend the keys sequentially, one-by-one, as needed.

As the first idea, we could have the signer time-stamp each signature, just as in BLT-TB. In case of a dispute, the signature with the earlier time-stamp would win and the later one would be considered a forgery. This would obviously make verification very difficult, but more importantly would give the signer a way to deny any signature: before signing a document d with a key z, the signer could use the same key to privately sign some dummy value x; when later demanded to honor the signature on d, the signer could show the signature on x and declare the signature on d a forgery.

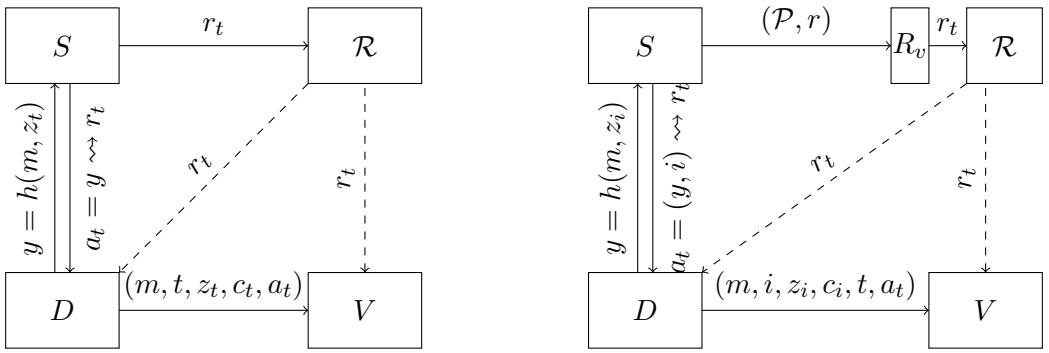

Figure 8.4: Interactions in BLT-TB (left) and BLT-BC (right): the signer uses the trusted device D to manage private keys; the server S aggregates client requests and posts round commitments to the repository $\mathcal{R}$, which both the signing device D and the verifier V query for the commitments; in BLT-BC, the repository has an additional validation layer R_v checking the proofs of correct operation accompanying the commitments before accepting them for inclusion in the repository.

To prevent this, we assign every signer to a designated server which allows each key to be used only once. A trivial solution would be to just trust the server to behave correctly. This would still not achieve non-repudiation, as the server could collect spent keys and create valid-looking signatures on behalf of the signer.

8.5.2 Description of the Scheme

Our proposed new scheme, which we will refer to as BLT-BC (for "blockchain"), consists of the following parties (Figure 8.4, right):

- The signer uses trusted device D to generate keys and then sign data, as in BLT-TB.
- The server S assists signers in generating signatures. S keeps a counter of spent keys for each signer and sends updates to the repository.
- The repository performs two tasks. The validation layer R_v verifies the correctness of each operation of S before accepting it and periodically commits the summary of current state to a public append-only storage layer $\mathcal{R}$.
- The verifier V is a relying party who verifies signatures, as in BLT-TB.

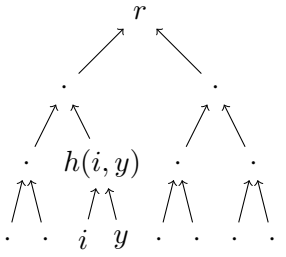

Figure 8.5: Server tree, showing the last key index i and the corresponding message authenticator y of the second client only.

The server maintains a hash tree with a dedicated leaf for each client (Figure 8.5). The value of the leaf is computed by hashing the pair (i, y) where i is the index of the last spent key and y is the last message received from the client (as detailed in **Signing** below).

Each public key must verifiably have just one leaf assigned to it. Otherwise, the server could set up multiple parallel counters for a client, increment only one of them in response to client requests, and use the others for forging signatures with keys the signer has already used and released.

One way to achieve that would be to have the server return the *shape* (that is, the directions to move to either the left or the right child on each step) of the path from the root of the tree to the assigned leaf when the client registers for service, and the client to include that shape when distributing its public key to verifiers. Another option would be to use the bits of the public key itself as the shape. Because most possible bit sequences are not actually used as keys, the hash tree would be a *sparse* one in this case.

Setup: Signer. To prepare to sign up to N messages, the signer:

1. Generates N unpredictable k-bit signing keys: $(z_1, \ldots, z_N) \leftarrow \mathcal{G}(N, 1^k)$.
2. Binds each key to its sequence number: $x_i \leftarrow h(i, z_i)$ for $i \in \{1, \ldots, N\}$.
3. Aggregates the key bindings into a hash tree: $p \leftarrow T^h(x_1, \ldots, x_N)$.
4. Registers the public key p with the server S.

The data structure giving the public key is similar to the one in the BLT-TB scheme (Figure 8.3), and also has the same purpose: to be able to extract the hash chains c_i linking the private key bindings to the public key: $h(i, z_i) \overset{c_i}{\rightsquigarrow} p$ for $i \in \{1, \ldots, N\}$.

Setup: Server. Upon receiving registration request from a signer, the server dedicates a leaf in its tree and sets i to 0 and y to an arbitrary value in that leaf.

Signing: Signer. Each signer keeps the index i of the next unused key z_i in its state. To sign message m, the signer:

1. Uses the current key to authenticate the message: $y \leftarrow h(m, z_i)$.
2. Sends the authenticator y to the server.
3. Waits for the server to return the hash chain a_t linking the pair (i, y) to the new published summary r_t: $h(i, y) \overset{a_t}{\rightsquigarrow} r_t$.
4. Checks that the shape of the received hash chain is correct and its output value matches the authentic r_t acquired directly from the repository.
5. If validation succeeds then outputs the tuple (i, z_i, c_i, t, a_t), where i is the key index, z_i is the i-th signing key, c_i is the hash chain linking the binding (i, z_i) to the public key p, and a_t is the hash chain linking (i, y) to the published r_t.
6. Increments its key counter: $i \leftarrow i + 1$.

Signing: Server. Upon receiving request y' from a signer, the server:

1. Extracts the hash chain a linking the current state of the client record (i, y) to the current root r of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
2. Updates the client's record from (i, y) to $(i' \leftarrow i + 1, y')$ and computes the corresponding new root hash r' of the server tree.
3. Submits the tuple (i, y, a, r, y', r') to the repository for validation and publishing.
4. Waits for the repository to end the round and publish r_t.
5. Uses the state of its hash tree corresponding to the published r_t to extract and return to all clients with pending requests the hash chains a_t linking their updated (i', y') records to the published r_t: $h(i', y') \overset{a_t}{\rightsquigarrow} r_t$.

Signing: Repository. The validation layer R_v of the repository $\mathcal{R}$ keeps as state the current value r^* of the root hash of the server tree. Upon receiving the update (i, y, a, r, y', r') from S, the validator verifies its correctness:

1. The claimed starting state of the server tree matches the current state of R_v: $r = r^*$.
2. The claimed starting state of the signer record agrees with the starting state of the server tree: $h(i, y) \overset{a}{\rightsquigarrow} r$.
3. The update of the client record increments the counter: $i' \leftarrow i + 1$.
4. The new state of the server tree corresponds to just this one change: $h(i', y') \overset{a}{\rightsquigarrow} r'$.
5. If all the above checks pass, R_v updates its own state accordingly: $r^* \leftarrow r'$.

Note that the hash chain a is the same in the verification of the starting state of the signer record against the starting state of the server tree and in the verification of the new state of the signer record against the new state of the server tree. This ensures no other leaves of the server tree can change with this update.

R_v operates in rounds. During a round, it receives updates from the server, validates them, and updates its own state accordingly. At the end of the round, it publishes the current value of its state as the new round commitment r_t in the append-only storage $\mathcal{R}$.

Verification. To verify that the message m and the signature $s = (i, z, c, t, a)$ match the public key p, the verifier:

1. Checks that z was committed as the i-th signing key: $h(i, z) \overset{c}{\rightsquigarrow} p$.
2. Retrieves the commitment r_t for the round t from repository $\mathcal{R}$.
3. Checks that the use of the key z to compute the message authenticator $y \leftarrow h(m, z)$ matches the key index i: $h(i, y) \overset{a}{\rightsquigarrow} r_t$.

Note that the signature is composed and sent to verifier only after the verification of r_t, which makes it safe for the signer to release the key z_i as part of the signature: the server has already incremented its counter i so that only z_{i+1} could be used to produce the next valid signature.

8.5.3 Implementation Considerations

Server-Supported Signing. The model of server-supported signing is a higher-level protocol not directly comparable to traditional signature algorithms like RSA. The model has some useful properties:

- It is possible to create a server-side log of all signing operations, so that in case of either actual or suspected key compromise there is a complete record, making damage control and forensics manageable.
- Key revocation can be implemented by setting the client's counter to some sentinel "infinite" value, and the server can also return a proof of this update after it has been committed to the repository.
- The server can add custom attributes, and even *trusted attributes* which can't be forged by the server itself: cryptographic time-stamp, address, policy ID, etc.

Finally, in scenarios where non-repudiation must be provided, all traditional schemes and algorithms must be supplemented with some server-provided functionalities like cryptographic time-stamping.

Blockchain-Backed Repository. The proposed scheme dictates that the repository must have the following properties:

- Updates are only accepted if their proof of correctness is valid.
- All commitments are final and immutable.
- Commitments are public, and their immutability is publicly verifiable.

To minimize trust requirements on the repository, we propose to re-use the patterns used for creating *blockchains*. We do not consider proof-of-work, focusing on byzantine fault tolerant state machine replication model.

Instead of full transactions, we record in the blockchain only aggregate hashes representing batches of transactions. This provides two benefits: (1) the size of the blockchain grows linearly in time, in contrast with the usual dependency on the number of transactions and storage size of transaction records; and (2) recording and publishing only aggregate hashes ensures privacy.

When implemented as a distributed robust public transaction ledger [GKL15], no single component of the repository needs to be trusted.

Scalable Architecture. Although presented above as a list of components, envisioned real-life deployment of the scheme is hierarchical, as shown in Figure 8.6.

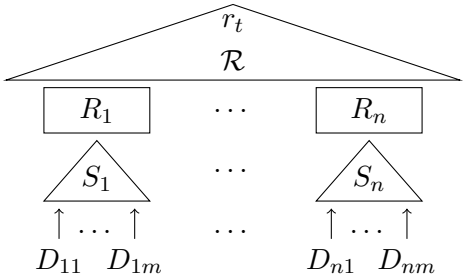

Figure 8.6: A scalable deployment architecture for BLT-BC: each client device D_{ij} is served by a designated server S_i audited by validation cluster R_i; round commitments from all validation clusters are aggregated into a meta-commitment by the repository $\mathcal{R}$.

The topmost layer is a distributed cluster of blockchain consensus nodes, each possibly operated by an independent "permissioned" party. The blockchain can accept inputs from multiple signing servers, each of which

may in turn serve many clients. Because of this hierarchical nature the scheme scales well performance-wise. In terms of the amount of data, as stated earlier, the size of blocks and the number of blocks does not depend on the number of clients and number of signatures issued.

8.5.4 Discussion

Performance. The efficiency of the new scheme for both signers and verifiers is at least on par with the state of the art.

The performance considerations for key generation and management on the client side are similar to the BLT-TB scheme (Section 8.4.2), except the number of private keys required is much smaller (assuming 10 signing operations per day, just $3\,650$ keys are needed for a year, compared to the 32 million keys in BLT-TB) and the effort required to generate and manage them, which was the main weakness of BLT-TB, is also correspondingly reduced.

Like in BLT-TB, the size of the signature in the new scheme is also dominated by the two hash chains. The key sequence membership proof contains $\log_2 N$ hash values, which is about 12 for the $3\,650$-element yearly sequence. The blockchain membership proof has $\log_2 K$ hash values, where K is the number of clients the service has. Even when the whole world (8 billion people) signs up, it's still only about 33 hash values. Conservatively assuming the use of 512-bit hash functions, the two hash chains add up to less than 3 kB in total.

Verification of the signature means re-computing the two hash chains and amounts to about 45 hash function evaluations.

Admittedly, the above estimates exclude the costs of querying the blockchain to acquire the committed r_t that both the signer and the verifier need. However, that is comparable to the need to access a time-stamping service when signing and an OCSP (Online Certificate Status Protocol) responder or a CRL (Certificate Revocation List) when verifying signatures in the traditional PKI (Public-Key Infrastructure) setup.

Security Model. The trusted repository $\mathcal{R}$ and its implications for the security model are similar to the BLT-TB scheme (Section 8.4.3).

In BLT-BC, the validation layer R_v is also critical for security, which obviously increases the trust base. As discussed in previous chapter for the repository $\mathcal{R}$, the security risks of introducing another trusted component into the system can be somewhat mitigated by implementing R_v as a distributed consensus cluster, but this adds additional engineering challenges and upkeep costs.

While the scheme also introduces state on the client side, this is mostly a usability concern and not a significant security risk. If a client loses track of its state and tries to re-use a key, the server will block the signing attempt.

8.6 Time-Stamped Scheme with One-Time Keys

Both the BLT-TB and BLT-BC signature schemes prevent other parties from misusing keys by making each key expire immediately after a legitimate use.

In BLT-BC, this is achieved by having a server track the first use of each key and prove the correctness of its operation through an auditing and publishing mechanism. Thus the security of BLT-BC rests on the reliability of the auditors and the publishing channel.

In BLT-TB, each key is explicitly bound to a time slot at the key-generation time and expires automatically when that time slot passes. The legitimate use of a key is proven by time-stamping the message-key pair at the correct time and the security of the scheme rests on the resilience of the time-stamping service against back-dating attacks, which is arguably a much smaller trust base.

8.6.1 Forward-Resistant Tags

Tag Systems. To combine the efficiency of the one-time keys of BLT-BC with the smaller trust base of BLT-TB, we can note that back-dating resistance of the time-stamp already prevents any attackers from moving the key usage events back in time. Thus, it would be sufficient for the key binding to only prevent it from being moved forward. Based on this observation, we introduce [BFL+19] the concept of *forward-resistant tags*.

Definition 22 (Tag system)**.** By a tag system we mean a triple $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ of algorithms, where:

- Gen is a probabilistic key-generation algorithm that, given as input the tag range T, produces a secret key sk and a public key pk.
- Tag is a tag-generation algorithm that, given as input the secret key sk and an integer $t \in \{1, \ldots, T\}$, produces a tag $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$.
- Ver is a verification algorithm that, given as input a tag τ, an integer t, and the public key pk, returns either 0 or 1, such that
$$\mathrm{Ver}(\mathrm{Tag}(\mathsf{sk}, t), t, \mathsf{pk}) = 1$$
whenever $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathrm{Gen}(T)$ and $1 \leq t \leq T$.

The above definition of a tag system is quite similar to that of a signature scheme consisting of procedures for key generation, signing, and verification. The fundamental difference is that a signature binds the use of the secret key to a message, while a tag binds the use of the secret key to a time.

Informally, in order to implement a forward resistant tag system, we have to bind each tag to a time t so that the tag can't be re-bound to a later time. As already mentioned, this notion could be seen as dual to time-stamping that prevents back-dating.

Induced Signature Scheme. We can now formalize the signature scheme induced by a tag system and a time-stamping repository.

Definition 23. A tag system $(\mathrm{Gen}, \mathrm{Tag}, \mathrm{Ver})$ and a time-stamping repository $\mathcal{R}$ induce a one-time signature scheme as follows:

- The signer $\mathcal{S}^{\mathcal{R}}(m)$ queries $t \leftarrow \mathcal{R}.\mathtt{time}$, creates $\tau \leftarrow \mathrm{Tag}(\mathsf{sk}, t)$, stores $\mathcal{R}.\mathtt{put}((m, \tau))$, and then returns (τ, t) as the signature.
- The verifier $\mathcal{V}^{\mathcal{R}}(m, (\tau, t), \mathsf{pk})$ queries $x \leftarrow \mathcal{R}.\mathtt{get}(t)$, and checks that $x = (m, \tau)$ and $\mathrm{Ver}(\mathsf{pk}, t, \tau) = 1$.

We use the simplistic model of the time-stamping service (omitting the aggregation layer) for convenience of formal analysis. A more refined model would make the security reductions really complex. For example, even for a seemingly trivial change, where $\mathcal{R}$ publishes a hash $h(m, \tau)$ instead of just (m, τ), one needs non-standard security assumptions about h such as non-malleability. In this section, we try to avoid these technicalities and focus on the basic logic of the tag-based signature scheme.

The BLT-TB Tag System. To simplify the analysis, we omit the aggregation of individual time-bound keys into a hash tree, and model the essence of the BLT-TB signature scheme as a tag system as follows:

- The secret key sk is a list $(z_1, \ldots, z_T)$ of T unpredictable values.
- The public key pk is the list $(f(z_1), \ldots, f(z_T))$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z_1, \ldots, z_T; t)$ outputs z_t.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key $(x_1, \ldots, x_T)$, checks that $1 \leq t \leq T$ and $f(\tau) = x_t$.

The BLT-OT Tag System. We now define the BLT-OT tag system (inspired by Lamport's one-time signatures [DH76]) as follows:

- The secret key sk is a list $(z_0, \ldots, z_{\ell-1})$ of $\ell = \lceil \log_2(T+1) \rceil$ unpredictable values.
- The public key pk is the list $(f(z_0), \ldots, f(z_{\ell-1}))$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z_0, \ldots, z_{\ell-1}; t)$ outputs an ordered subset $(z_{j_1}, \ldots, z_{j_m})$ of components of the secret key sk such that $0 \leq j_1 < \ldots < j_m \leq \ell - 1$ and $2^{j_1} + \ldots + 2^{j_m} = t$.
- The verification algorithm Ver, given as input a sequence $(z_{j_1}, \ldots, z_{j_m})$, an integer t, and the public key $(x_0, \ldots, x_{\ell-1})$, checks that:
  1. $f(z_{j_1}) = x_{j_1}, \ldots, f(z_{j_m}) = x_{j_m}$; and
  2. $0 \leq j_1 < \ldots < j_m \leq \ell - 1$; and
  3. $2^{j_1} + \ldots + 2^{j_m} = t$; and
  4. $1 \leq t \leq T$.

The BLT-W Tag System. We now define the BLT-W tag system (inspired by Winternitz's idea [Mer87] for optimizing the size of Lamport's one-time signatures) as follows:

- The secret key sk is an unpredictable value z.
- The public key pk is $f^T(z)$, where f is a one-way function.
- The tagging algorithm $\mathrm{Tag}(z; t)$ outputs the value $f^{T-t}(z)$.
- The verification algorithm Ver, given as input a tag τ, an integer t, and the public key x, checks that $1 \leq t \leq T$ and $f^t(\tau) = x$.

8.6.2 Description of the Scheme

The signature scheme induced by the BLT-OT tag system according to Definition 23 would come with the requirement that the signer must know in advance the time at which its request reaches the time-stamping service. This is hard to achieve in practice, in particular for personal signing devices such as smart cards that lack built-in clocks. To overcome this limitation, we construct the BLT-OT one-time signature scheme as follows.

Key Generation. Let ℓ be the number of bits that can represent any time value t when the signature may be created (e.g. $\ell = 32$ for POSIX time up to year 2106). The private key is generated as $\mathsf{sk} = (z_0, \ldots, z_{\ell-1})$, where z_i are unpredictable values, and the public key as $\mathsf{pk} = h(X)$, $X = (x_0, \ldots, x_{\ell-1})$, $x_i = f(z_i)$, where h is a second pre-image resistant hash function and f a one-way function.

Public Key Distribution. To aid instant key revocation, also the identity ID_c of the client and the identity ID_s of the designated time-stamping service should be distributed along with the public key (within the public key certificate in a typical PKI-like setup). Upon receiving a revocation notice, the service stops serving the affected client, and thus it is not possible to generate signatures using revoked keys.

Signing. To sign a message m, the client:

1. Gets a time-stamp S_t on the record (m, X, ID_c) from the time-stamping service designated by ID_s.
2. Extracts the ℓ-bit time value t from S_t and creates the list $W = (w_0, \ldots, w_{\ell-1})$, where
   - $w_i = z_i$ if the i-th bit of t is 1, or
   - $w_i = x_i = f(z_i)$ otherwise.
3. Disposes of the private key $(z_0, \ldots, z_{\ell-1})$ to prevent its re-use.
4. Emits (W, S_t) as the signature.

Verification. To verify the signature (W, S_t) on the message m against $(\mathsf{pk}, ID_c, ID_s)$, the verifier:

1. Extracts time t from the time-stamp S_t.
2. Recovers the list $X = (x_0, \ldots, x_{\ell-1})$ by computing
   - $x_i = f(w_i)$ if the i-th bit of t is 1, or
   - $x_i = w_i$ otherwise.
3. Checks that the computed X matches the public key: $h(X) = \mathsf{pk}$.
4. Checks that S_t is a valid time-stamp issued at time t by service ID_s on the record (m, X, ID_c).

Using the reduction techniques from previous sections to formally prove the security of this optimized signature scheme is complicated by both the iterated use of f and the more abstract view of the time-stamping service, and is left as future work.

8.6.3 Discussion

The BLT-TB scheme works well for powerful devices that are constantly running and have reliable clocks. These are not reasonable assumptions for personal signing devices such as smart cards, which have very limited capabilities and are not used very often. Generating keys could take hours or even days of non-stop computing on such devices. This is clearly impractical, and also wasteful as most of the keys would go unused.

The BLT-OT scheme proposed in Section 8.6.2 solves these problems at the cost of introducing state on the client side. As the scheme is targeted towards personal signing devices, the statefulness is not a big risk, because these devices are not backed up and also do not support parallel processing. The benefit in addition to improved efficiency is that the device no longer needs to know the current time while preparing a signing request. Instead, it can just use the time from the time-stamp when composing the signature.

Table 8.1: Efficiency of hash-based one-time signature schemes. We assume 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call. BLT-W refers to a variation of the scheme with Winternitz-inspired optimization, as described in Section 8.6.3

Scheme	Key gen. time	Sig. time	Ver. time	Sig. size
Lamport	1 025	1 024	513	256
Winternitz ($w = 4$)	1 089	1 088	1 021	68
BLT-OT	65	64 + TS	33 + 33	32 + 33
BLT-W ($w = 2$)	65	64 + TS	49 + 33	16 + 33

Performance as One-Time Scheme. When implemented as described in Section 8.6.2, the cost of generating a BLT-OT key pair is ℓ random key generations and $\ell + 1$ hashing operations, the cost of signing $\ell + 1$ hashing operations and one time-stamping service call, and the cost of signature verification at most $\ell + 1$ hashing operations and one time-stamp verification. In this case the private key would consist of ℓ one-time keys and the public key of one hash value, and the signature would contain ℓ hash values and one time-stamp token. The private storage size can be optimized by generating the one-time keys from one true random seed using a pseudo-random generator. Then the cost of signing increases by ℓ operations, as the one-time keys would have to be re-generated from the seed before signing. This version is listed as BLT-OT in Table 8.1.

Winternitz's idea [Mer87] for optimizing the size of Lamport's one-time signatures [DH76] can also be applied to BLT-OT. Instead of using one-step hash chains $z_i \to h(z_i) = x_i$ to encode single bits of t, we can use longer chains $z_i \to h(z_i) \to \ldots \to h^n(z_i) = x_i$ and publish the value $h^{n-j}(z_i)$ in the signature to encode the value j of a group of bits of t. When encoding groups of w bits of t in this manner, the chains have to be $n = 2^w$ steps long. This reduces the size of the signature by w times, but increases the costs of key generation and signing by a bit less than $\frac{2^w-1}{w}$ times and the cost of verification by a bit less than $\frac{2^w-1}{w}$ times. Note that for $w = 2$, only the verification cost increases by about 50%! Also note that in contrast to applying this idea to Lamport's signatures, in BLT-W no additional countermeasures are needed to prevent an adversary from stepping the hash chains forward: the time in the time-stamp takes that role. This version is listed as BLT-W in Table 8.1.

To compare BLT-OT signature sizes and verification times to other schemes, we also need to estimate the size of hash-trees built by the time-stamping service. Even assuming the whole world (8 billion people) will use the time-stamping service in every aggregation round, an aggregation tree of 33 layers will suffice. We also assume that in all schemes one-time private keys will be generated on-demand from a single random seed and public keys will be aggregated into a single hash value. Therefore, the key sizes will be the same for all schemes and are not listed in Table 8.1.

Table 8.2: Efficiency of hash-based many-time signature schemes. We assume key supply for at least 3 650 signatures, 256-bit hash functions, 32-bit time values, and time-stamping hash-tree with 33 levels. Times are in hashing operations and signature sizes in hash values. TS in BLT schemes stands for the time-stamping service call.

Scheme	Key gen. time	Sig. time	Ver. time	Sig. size
XMSS	897 024	8 574	1 151	79
SPHINCS	ca 16 000	ca 250 000	ca 7 000	ca 1 200
BLT-TB	ca 96 000 000	50 + TS	25 + 33	25 + 33
BLT-OT-N	240 900	64 + TS	45 + 33	44 + 33
BLT-W-N ($w = 2$)	240 900	64 + TS	61 + 33	28 + 33

Performance as Many-Time Scheme. A one-time signature scheme is not practical by itself. Merkle [Mer79] proposed aggregating multiple public keys of a one-time scheme using a hash tree to produce so-called N-time schemes. Assuming 10 signing operations per day, a set of 3 650 BLT-OT keys would be sufficient for a year. The key generation costs would obviously grow correspondingly. The change in signing time would depend on how the hash tree would be handled. If sufficient memory is available to keep the tree (which does not contain private key material and thus may be stored in regular memory), the authenticating hash chains for individual one-time public keys could be extracted with no extra hash computations. Signature size and verification time would increase by the 12 additional hashing steps linking the one-time public keys to the root of the aggregation tree. This scheme is listed as BLT-OT-N in Table 8.2, where we compare it with the following schemes:

- XMSS is a stateful scheme like BLT-OT-N; the values in Table 8.2 are computed by taking $N = 2^{12} = 4\,096$ and leaving other parameters as in [BDH11];

- SPHINCS is a stateless scheme and can produce an indefinite number of signatures; the values in Table 8.2 are inferred from [BHH+15] counting invocations of the ChaCha12 cipher on 64-byte inputs as equivalent to hash function evaluations;
- The values for BLT-TB in Table 8.2 are inferred from Section 8.4.3.

Security Model. Like those of BLT-TB and BLT-BC, the security model of BLT-OT also relies on the repository $\mathcal{R}$ for the unforgeability of the signatures. This is in contrast with XMSS and SPHINCS, both traditional "standalone" signature schemes.

Unlike BLT-BC, management of client-side state to track spent keys is a real security concern for BLT-OT. Indeed, even using a private key just twice may in the worst case leak all key components and give an adversary an easy opportunity to forge signatures on any chosen messages at any chosen time in the future. Although this is not necessary with correct private key management, an external service similar to the validation layer of BLT-BC may in fact provide a useful safety net to reduce this risk. Such state management concerns also apply to XMSS, while SPHINCS, being a stateless scheme, is not affected.

A weakness of the BLT family compared to XMSS and SPHINCS is the higher requirements that the BLT schemes place on the underlying hash function. The unforgeability proof for BLT-TB signature scheme in Chapter 8.4 assumes the hash function models a random oracle, which is a very high bar.

The forward-resistance proofs of BLT-TB and BLT-OT tag systems given in Section 8.6.1 only assume one-wayness from the underlying hash function, but these proofs cover only a small part of the whole signature scheme. Extending the security proofs to complete signature schemes while keeping the assumptions minimal is likely to require changes in the definitions of the schemes.

For example, the plain hash trees aggregating the one-time or time-bound key pairs into a many-time key set will likely have to be replaced by more complicated constructs, like has already been done in XMSS and SPHINCS. How such hardening will affect the performance of the signature schemes remains to be determined by future research.

8.7 Conclusions and Outlook

We have proposed several hash-based server-assisted digital signature schemes. A novel design element of the schemes is their reliance on time-stamping service as an inherent component. The performance of the new schemes is very competitive, as indicated in Tables 8.1 and 8.2, but the reliance on time-stamping service adds dependence on a security-critical external component.

The BLT-TB scheme described in Section 8.4 is suitable for use in server applications that need to produce a lot of signatures. The scheme features efficient signing and verification and small signatures. Only the key generation is quite expensive, but still tolerable on full-sized computers. The scheme also requires the signer to have a reliable clock and direct network access, which are reasonable assumptions for servers. A benefit is that the scheme is stateless in the sense that the key to be used is determined by the signing time and thus the signer does not need to explicitly track spent keys and also access to the private key does not need to be synchronized in a parallel execution environment.

In contrast, the BLT-BC and BLT-OT schemes described in Section 8.5 and Section 8.6, respectively, are suitable for personal signing devices that are used only occasionally. In addition to small signatures and efficient signing and verification, also key generation costs are relatively low. The price for this efficiency is the introduction of state on the client side. However, as typically personal key management devices, such as smart cards and USB tokens, are not backed up and do not support multi-processing, the risks related to managing key state are significantly reduced.

In the BLT-BC scheme, the correct management of the client state is additionally enforced by a supporting server. In particular, the composition of the authenticated data structure and the blockchain-backed consensus-based auditing layer described in Section 8.5 may be of independent interest.

A weakness of the formal analysis of the tag-based schemes in Section 8.6, in particular compared to the analysis in Section 8.4, is the simplistic modeling of the time-stamping service. Some progress towards addressing this issue has been made in [BFLT20] and further work is ongoing.

Another limitation of the present work is that the formal security reductions are done in classical setting and post-quantum security of the schemes is supported only indirectly, by referring to the quantum resilience of hash functions in general.

Formal analysis in quantum setting is hindered, among other difficulties, by the fact that our signature schemes depend on time-stamping and there is currently no well-defined notion of back-dating resistance of time-stamping in quantum setting. There is hope, however, that the collapsing property [Unr16b, Unr16a, CBH+18] of hash functions could be useful in moving to quantum setting.

Chapter 9

Conclusion

This document presented recent advances of PRIViLEDGE partners in designing privacy-preserving cryptographic primitives for distributed ledgers. The formal security notions for some of these cryptographic primitives have been illustrated by PRIViLEDGE partners in the deliverable D2.2. The partners are continuing their work towards finding both theoretical and real-world applications of these results and improving the constructions presented here. Follow up results will appear in deliverable D2.4.

Bibliography

[AAN18] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. Practically efficient secure distributed exponentiation without bit-decomposition. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 291–309. Springer, 2018.

[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 Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[ABC+98] Ross J. Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger M. Needham. A new family of authentication protocols. *Operating Systems Review*, 32(4):9–20, 1998.

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 2002.

[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.

[ADMM16] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. *Commun. ACM*, 59(4):76–84, 2016.

[Ank52] N. C. Ankeny. The least quadratic non residue. *Annals of Mathematics*, 55(1):65–72, 1952.

[ANPS07] Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In *ASIACRYPT 2007, Proceedings*, volume 4833 of *LNCS*, pages 130–146. Springer, 2007.

[AS11] Elena Andreeva and Martijn Stam. The symbiosis between collision and preimage resistance. In *IMACC 2011, Proceedings*, volume 7089 of *LNCS*, pages 152–171. Springer, 2011.

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 106–115. IEEE Computer Society, 2001.

[BAZB19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptology ePrint Archive*, 2019.

[BB08] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *J. Cryptology*, 21(2):149–177, 2008.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2018.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 757–788, 2018.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors. *Post-Quantum Cryptography*. Springer, 2009.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT*, volume 2248 of *LNCS*, pages 566–582. Springer, 2001.

[BCC+09] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, pages 108–125, 2009.

[BCC+14] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive, Report 2014/571, 2014. `http://eprint.iacr.org/2014/571`.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *EUROCRYPT*, 2016.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 90–108, 2013.

[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 *Proceedings of the IEEE Symposium on Security & Privacy*, 2014.

[BCG+17] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *Proceedings of Asiacrypt 2017*, 2017.

[BCG+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune K. Jakobsen, and Mary Maller. Nearly linear-time zero-knowledge proofs for correct program execution. *IACR Cryptology ePrint Archive*, 2018:380, 2018.

[BCGV16] Eli Ben-Sasson, Alessandro Chiesa, Ariel Gabizon, and Madars Virza. Quasi-linear size zero knowledge from linear-algebraic pcps. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 33–64, 2016.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 781–796, 2014.

[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.

[BFG13] David Bernhard, Georg Fuchsbauer, and Essam Ghadafi. Efficient signatures of knowledge and DAA in the standard model. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, pages 518–533, 2013.

[BFL+19] 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.

[BFLT20] Ahto Buldas, Denis Firsov, Risto Laanoja, and Ahto Truu. Verified security of BLT signature scheme. In *ACM SIGPLAN CPP 2020, Proceedings*, pages 244–257. ACM, 2020.

[BFS16] Mihir Bellare, Georg Fuchsbauer, and Alessandra Scafuro. NIZKs with an untrusted CRS: security in the face of parameter subversion. In *ASIACRYPT*, pages 777–804, 2016.

[BFVV19] Vincenzo Botta, Daniele Friolo, Daniele Venturi, and Ivan Visconti. The rush dilemma: Attacking and repairing smart contracts on forking blockchains. *IACR Cryptol. ePrint Arch.*, 2019:891, 2019.

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, pages 263–280, 2012.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Financial Cryptography Workshops*, volume 10958 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2018.

[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.

[BGM+18] Christian Badertscher, Juan A. Garay, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. But why does it work? A rational protocol design treatment of bitcoin. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 34–65, 2018.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988.

[BGZ16a] Iddo Bentov, Ariel Gabizon, and David Zuckerman. Bitcoin beacon. *CoRR*, abs/1605.04559, 2016.

[BGZ16b] Iddo Bentov, Ariel Gabizon, and David Zuckerman. Bitcoin beacon. *CoRR*, abs/1605.04559, 2016.

[BH84] Duncan A Buell and Richard H Hudson. On runs of consecutive quadratic residues and quadratic nonresidues. *BIT Numerical Mathematics*, 24(2):243–247, 1984.

[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.

[BHT98] Gilles Brassard, Peter Hoyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN'98, Proceedings*, volume 1380 of *LNCS*, pages 163–169. Springer, 1998.

[BIB89] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209, 1989.

[BK14a] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. *IACR Cryptology ePrint Archive*, 2014:129, 2014.

[BK14b] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, pages 421–439, 2014.

[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.

[BL95] W. Bosma and H.W. Lenstra. Complete systems of two addition laws for elliptic curves. *Journal of Number Theory*, 53(2):229 – 240, 1995.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.

[BL13] Ahto Buldas and Risto Laanoja. Security proofs for hash tree time-stamping using hash functions with small output size. In *ACISP 2013, Proceedings*, volume 7959 of *LNCS*, pages 235–250. Springer, 2013.

[BLLT14] Ahto Buldas, Risto Laanoja, Peeter Laud, and Ahto Truu. Bounded pre-image awareness and the security of hash-tree keyless signatures. In *ProvSec 2014, Proceedings*, volume 8782 of *LNCS*, pages 130–145. Springer, 2014.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004.

[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 *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 324–356, 2017.

[BN10] Ahto Buldas and Margus Niitsoo. Optimally tight security proofs for hash-then-publish time-stamping. In *ACISP 2010, Proceedings*, volume 6168 of *LNCS*, pages 318–335. Springer, 2010.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, pages 409–426, 2006.

[BS04] Ahto Buldas and Märt Saarepera. On provably secure time-stamping schemes. In *ASIACRYPT 2004, Proceedings*, volume 3329 of *LNCS*, pages 500–514. Springer, 2004.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[Bur63] DA Burgess. A note on the distribution of residues and non-residues. *Journal of the London Mathematical Society*, 1(1):253–256, 1963.

[BV07] Johannes A. Buchmann and Ulrich Vollmer. *Binary quadratic forms - an algorithmic approach*, volume 20 of *Algorithms and computation in mathematics*. Springer, 2007.

[BW88] Johannes A. Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *J. Cryptology*, 1(2):107–118, 1988.

[Can01a] 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.

[Can01b] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science*. IEEE, 2001.

[Can18] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology eprint archive, report 2000/067, December 2018.

[CBH+18] Jan Czajkowski, Leon Groot Bruinderink, Andreas Hülsing, Christian Schaffner, and Dominique Unruh. Post-quantum security of the sponge construction. In *PQCrypto 2018, Proceedings*, volume 10786 of *LNCS*, pages 185–204. Springer, 2018.

[CCas08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *Advances in Cryptology — ASIACRYPT*, volume 5350 of *LNCS*, pages 234–252. Springer, 2008.

[CCFG16] Pyrros Chaidos, Véronique Cortier, Georg Fuchsbauer, and David Galindo. Beleniosrf: A non-interactive receipt-free electronic voting scheme. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1614–1625, 2016.

[CDHK15a] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT*, volume 9453 of *LNCS*, pages 262–288. Springer, 2015.

[CDHK15b] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*, pages 262–288, 2015.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil Vadhan, editor, *Theory of Cryptography*, volume 4392 of *LNCS*, pages 61–85. Springer, 2007.

[CDT19] Jan Camenisch, Manu Drijvers, and Björn Tackmann. Multi-protocol UC and its use for building modular and efficient protocols. Cryptology eprint archive, report 2019/065, January 2019.

[CEK+16] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT*, volume 10032 of *LNCS*, pages 807–840. Springer, 2016.

[CG08] Jan Camenisch and Thomas Groß. Efficient attributes for anonymous credentials. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 345–356, 2008.

[CGJ19] Arka Rai Choudhuri, Vipul Goyal, and Abhishek Jain. Founding secure computation on blockchains. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 2019.

[CH02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT*, volume 2332 of *LNCS*, pages 337–351. Springer, 2002.

[CKLM13] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Succinct malleable nizks and an application to compact shuffles. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 100–119, 2013.

[CKLM14] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable signatures: New definitions and delegatable anonymous credentials. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 199–213, 2014.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, pages 78–96, 2006.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *ACM STOC*, pages 364–369, 1986.

[CLT18] Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. Practical fully secure unrestricted inner product functional encryption modulo p. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 733–764. Springer, 2018.

[Cor05] Luis Carlos Coronado García. *Provably Secure and Practical Signature Schemes*. PhD thesis, Darmstadt University of Technology, Germany, 2005.

[COSV17] Michele Ciampi, Rafail Ostrovsky, Luisa Siniscalchi, and Ivan Visconti. Delayed-input non-malleable zero knowledge and multi-party coin tossing in four rounds. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 711–742. Springer, 2017.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burton S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO*, volume 1294 of *LNCS*, pages 410–424. Springer, 1997.

[CS03] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2010.

[CZJ+17] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via PVORM. In *ACM CCS*, pages 701–717. ACM, 2017.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[DH79] Whitfield Diffie and Martin E. Hellman. Privacy and authentication: An introduction to cryptography. *Proc. IEEE*, 67(3):397–427, 1979.

[Dia88] Persi Diaconis. Group representations in probability and statistics. *Lecture notes-monograph series*, 11:i–192, 1988.

[Dix08] John D. Dixon. Generating random elements in finite groups. *Electron. J. Comb.*, 15(1), 2008.

[DNT12] Morten Dahl, Chao Ning, and Tomas Toft. On secure two-party integer division. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers*, volume 7397 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2012.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography — PKC*, volume 3386 of *LNCS*, pages 416–431. Springer, 2005.

[Edw07] Harold M. Edwards. A normal form for elliptic curves. *BulletinAmerican Mathematical Society*, 2007.

[ElG85a] Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[ElG85b] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[ER65] Paul Erdös and Alfréd Rényi. Probabilistic methods in group theory. *Journal d'Analyse Mathématique*, 14(1):127–138, 1965.

[Eth] Ethereum. `https://www.ethereum.org/`.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, pages 33–62, 2018.

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume I*, pages 308–317. IEEE Computer Society, 1990.

[FMMO18] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018.

[FPS+18] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. Practical accountability of secret processes. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 657–674, 2018.

[Gei10] Martin Geisler. Cryptographic protocols: theory and implementation. *PhD thesis, University of Aarhus*, 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.

[GG19] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. *IACR Cryptology ePrint Archive*, 2019:114, 2019.

[GGM14] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[GGM16] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, volume 9603 of *LNCS*, pages 81–98. Springer, 2016.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, pages 626–645, 2013.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

[GJKR07] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310, 2015.

[GKM+18a] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *CRYPTO (3)*, volume 10993 of *Lecture Notes in Computer Science*, pages 698–728. Springer, 2018.

[GKM+18b] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 698–728. Springer, 2018.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 473–489, 2017.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ron Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[GOS12a] Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM*, 59(3), June 2012.

[GOS12b] Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *J. ACM*, 59(3):11:1–11:35, 2012.

[GPS08] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *28th ACM STOC, Proceedings*, pages 212–219. ACM, 1996.

[Gro15] Jens Groth. Efficient fully structure-preserving signatures for large messages. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology — ASIACRYPT*, volume 9452 of *LNCS*, pages 239–259. Springer, 2015.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, pages 305–326, 2016.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel Smart, editor, *Advances in Cryptology — EUROCRYPT*, volume 4965 of *LNCS*, pages 415–432. Springer, 2008.

[GS09] Vipul Goyal and Amit Sahai. Resettably secure computation. In *EUROCRYPT*, pages 54–71, 2009.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, 2011.

[HM16] Derek Holt and Henning Makholm. What are some Group representation of the rubik's cube group? StackExchange Mathematics `https://math.stackexchange.com/questions/1587307/what-are-some-group-representation-of-the-rubiks-cube-group`, 2016. [Online; accessed 23-January-2020].

[Hoo12] S.J.A. Hoogh, de. *Design of large scale applications of secure multiparty computation : secure linear programming*. PhD thesis, Department of Mathematics and Computer Science, 2012.

[Hum03] Patrick Hummel. On consecutive quadratic non-residues: a conjecture of issai schur. *Journal of Number Theory*, 103(2):257–266, 2003.

[HWCD08] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2008.

[Hyp] Hyperledger Fabric Maintainers. Hyperledger Fabric pluggable endorsement and validation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html.

[JMV01] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.

[KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *ACM CCS*, pages 30–41, 2014.

[KB16a] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 543–571, 2016.

[KB16b] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 543–571, 2016.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 839–858, 2016.

[Kob87] Neal Koblitz, editor. *Elliptic curve cryptosystems*, volume 5350 of *Mathematics of Computation 48*. American Mathematical Society, 1987.

[KP15] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptology ePrint Archive*, 2015:1019, 2015.

[Kra15] Vlad Krasnov. Go crypto: bridging the performance gap. https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/, May 2015.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 177–194. Springer, 2010.

[Lam81] Leslie Lamport. Password authentification with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[Lin03] Yehuda Lindell. Parallel coin-tossing and constant-round secure two-party computation. *J. Cryptology*, 16(3):143–184, 2003.

[Lon19] Lipa Long. Binary quadratic forms. GitHub `https://github.com/Chia-Network/vdf-competition/blob/master/classgroups.pdf`, 2019. [Online; accessed 23-January-2020].

[LS90] Dror Lapidot and Adi Shamir. Publicly verifiable non-interactive zero-knowledge proofs. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 353–365. Springer, 1990.

[Luk05] András Lukács. Generating random elements of abelian groups. *Random Struct. Algorithms*, 26(4):437–445, 2005.

[Max13] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. `bitcointalk.org`, August 2013.

[MBKM] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. `https://eprint.iacr.org/2019/099`.

[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.

[Mer79] Ralph C. Merkle. *Secrecy, Authentication and Public Key Systems*. PhD thesis, Stanford University, 1979.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.

[MKF+16] David A. McGrew, Panos Kampanakis, Scott R. Fluhrer, Stefan-Lukas Gazdag, Denis Butin, and Johannes A. Buchmann. State management for hash-based signatures. In *SSR 2016, Proceedings*, volume 10074 of *LNCS*, pages 244–260. Springer, 2016.

[MLLL+12] Enrique Martín-López, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L. O'Brien. Experimental realization of Shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773–776, 2012.

[MNP01] Andreas Meyer, Stefan Neis, and Thomas Pfahler. First implementation of cryptographic protocols based on algebraic number fields. In Vijay Varadharajan and Yi Mu, editors, *Information Security and Privacy, 6th Australasian Conference, ACISP 2001, Sydney, Australia, July 11-13, 2001, Proceedings*, volume 2119 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2001.

[MS18] Izaak Meckler and Evan Shapiro. Coda: Decentralized cryptocurrency at scale, 2018.

[Nak08a] Satoshi Nakamoto. Bitcoin: A peer-to-peer electionic cash system. unpublished, 2008., 2008.

[Nak08b] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[NIS01] NIST. Secure hash standard (SHS). FIPS 180-4, 2001.

[NIS16] NIST. Post-quantum cryptography. `https://www.nist.gov/pqcrypto`, 2016.

[NVV18] Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In *Symposium on Networked Systems Design and Implementation*, pages 65–80. USENIX, 2018.

[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 Sata, editors, *Financial Cryptography and Data Security*, volume 10958 of *LNCS*, pages 43–63. Springer, 2018.

[PCTS02] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Song. The TESLA broadcast authentication protocol. *CryptoBytes*, 5(2):2–13, 2002.

[Ped91a] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 129–140, 1991.

[Ped91b] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO*, volume 576 of *LNCS*, pages 129–140. Springer, 1991.

[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.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Proceedings of the Cryptographers Track at the RSA Conference*, volume 9610 of *LNCS*, pages 111–126. Springer, 2016.

[PS18] David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel Smart, editor, *Topics in Cryptology — CT-RSA*, volume 10808 of *LNCS*, pages 319–338. Springer, 2018.

[PSS17] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *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*, pages 643–673, 2017.

[PST13] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 222–242, 2013.

[quo] Quorum. https://www.goquorum.com/.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 403–428. Springer, 2016.

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *22nd ACM STOC, Proceedings*, pages 387–394. ACM, 1990.

[RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE 2004, Revised Papers*, volume 3017 of *LNCS*, pages 371–388. Springer, 2004.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[SA19] Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol: With an application to mixnets. *IACR Cryptology ePrint Archive*, 2019:768, 2019.

[SABB+18] Alberto Sonnino, Mustafa Al-Bassam, Sherar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. arXiv:1802.07344, August 2018.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan 1991.

[Sch03] Daniel Schielzeth. Realisierung der elgamal-verschlüsselung in quadratischen zählkorpern (master's thesis). Technische Universität Berlin `http://www.math.tu-berlin.de/~kant/publications.html`, 2003.

[Sch17] Berry Schoenmakers. Explicit optimal binary pebbling for one-way hash chain reversal. In *FC 2016, Revised Selected Papers*, volume 9603 of *LNCS*, pages 299–320. Springer, 2017.

[Sch18] Berry Schoenmakers. MPyC secure multiparty computation in Python. GitHub `https://github.com/lschoe/mpyc`, 2018.

[Ser77] Jean-Pierre Serre. *Linear representations of finite groups*, volume 42 of *Graduate texts in mathematics*. Springer, 1977.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

[Son] Sonic reference implementation. `https://github.com/zknuckles/sonic`.

[SSV19] Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Publicly verifiable proofs from blockchains. In Dongdai Lin and Kazue Sako, editors, *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part I*, volume 11442 of *Lecture Notes in Computer Science*, pages 374–401. Springer, 2019.

[SSV20] Alessandra Scafuro, Luisa Siniscalchi, and Ivan Visconti. Publicly verifiable zero knowledge from (collapsing) blockchains. *IACR Cryptol. ePrint Arch.*, 2020:1435, 2020.

[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.

[Szy04] Michael Szydlo. Merkle tree traversal in log space and time. In *EUROCRYPT 2004, Proceedings*, volume 3027 of *LNCS*, pages 541–554. Springer, 2004.

[Tof07] T. Toft. *Primitives and applications for multi-party computation*. PhD thesis, Aarhus University, 2007.

[Unr16a] Dominique Unruh. Collapse-binding quantum commitments without random oracles. In *ASIACRYPT 2016, Proceedings, Part II*, volume 10032 of *LNCS*, pages 166–195. Springer, 2016.

[Unr16b] Dominique Unruh. Computationally binding quantum commitments. In *EUROCRYPT 2016, Proceedings, Part II*, volume 9666 of *LNCS*, pages 497–527. Springer, 2016.

[vS13] Nicolas van Saberhagen. CryptoNote v 2.0. `https://cryptonote.org/whitepaper.pdf`, October 2013.

[Wil16] Zooko Wilcox. The design of the ceremony. `https://z.cash/blog/the-design-of-the-ceremony.html`, October 2016.

[Wor18] ZKProof Standards Workshop, 2018. https://zkproof.org/proceedings-snapshots/zkproof-implementation-20180801.pdf.

[ZCa] ZCash. `https://z.cash/`.