DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D3.3 – Revision of Extended Core Protocols

Due date of deliverable: 30 June 2021
Actual submission date: 29 June 2021

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

Lead contractor: Guardtime AS
Duration: 36 months

 Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020	
Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D3.3

Revision of Extended Core Protocols

Editors
Michele Ciampi (UEDIN)
Aikaterini-Panagiota Stouka (UEDIN)
Thomas Zacharias (UEDIN)

Contributors
Daniele Friolo (UNISA)
Vincenzo Iovino (UNISA)
Ivan Visconti (UNISA)
Aggelos Kiayias (UEDIN)
Volkhov Misha (UEDIN)
Markulf Kohlweiss (UEDIN)
Toon Segers (TUE)
Marko Vukolic (IBM)
Ahto Truu (GT)

Reviewers
Risto Laanoja (GT)
Toon Segers (TUE)

29 June 2021
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.

Contents

Chapter 1

Executive Summary

This deliverable presents the revisions made to the core protocols presented in D3.2 based on the feedback and experience with the WP1 use cases. This deliverable also contains new results that do not directly extend results presented in D3.2 and results of independent interest not strictly related with to Use Cases and the Toolkits.

The contributions presented in the following chapters include both peer-reviewed and ongoing research work. We give a brief overview of the deliverable's content, specifying what contribution is based on the feedback and experience with the WP1. Where it is relevant, we also summarize the impact external to the project (in terms of scientific publications) of each contribution.

Updatable Blockchains. Chapter 2 proposes the first formal treatment of decentralized software updates for blockchain systems. In particular, we provide a definition of *secure update* for blockchain systems, and propose two generic compilers that take a blockchain system, and turn it into a new blockchain system that tolerates updates. The chapter only deals with the aspect related to how the update should be performed in a secure way, assuming that the nodes that are participating in the system have already agreed on the code of the new blockchain system (i.e., the code to which the blockchain should be updated to). For a comprehensive discussion on how the nodes of the system can reach an agreement on the updated code we refer the reader to Chapter 3 of the Deliverable D4.2. Part of the results have been collected in a paper presented to Esorics 2020 [CKKZ20] and have been used for the development of the Use Case 4.

Mining for Privacy. Chapter 3 studies the setup assumptions of succinct non-interactive zero-knowledge arguments (zk-SNARKS), and investigate whether the process of generating the setup can be decentralized using a blockchain. In particular, in this chapter we investigate whether the nodes of a blockchain system can be incentivized in generating the setup required for a zk-SNARKS, without adding additional assumptions on the honesty of the nodes that are participating in the system. Part of the results presented in this chapter have been presented to Financial Cryptography and Data Security 2021 [KKK21].

Mir-BFT: High-Throughput Robust BFT for Decentralized Networks In Chapter 4, we describe Mir-BFT, a basis for the flexible consensus toolkit in Hyperledger Fabric. Mir-BFT is a robust Byzantine fault-tolerant (BFT) total order broadcast protocol aimed at maximizing throughput on wide-area networks (WANs), targeting deployments in decentralized networks, such as permissioned and Proof-of-Stake permissionless blockchain systems. Mir-BFT is developed open-source and is planned to be integrated in a major permissioned blockchain project. We also evaluate Mir-BFT under different crash and Byzantine faults, demonstrating its performance robustness.

Removing Data from Bitcoin. In Chapter 5 we study the problem of removing illegal content that might appear on Bitcoin blockchain. We first provide a theoretical solution to sanitize Bitcoin blockchain from illicit content showing how to tackle the two main ways that Bitcoin provides to upload data on a blockchain (i.e., coinbase transaction and OP_RETURN). We then show an experimental validation of our approach describing the use of a tool that we have implemented and that can be effectively used to sanitize the Bitcoin blockchain.

Contact Tracing. In Chapter 6 we investigate how blockchain technology enables better systems for contact tracing. In particular, we present a contact tracing system that is more resilient to mass surveillance attacks compared to many existing solutions. The system is based on using a bulletin board, implementable through a blockchain, to allow anonymous notifications of at-risk contacts. In particular, we use a blockchain as shared memory so that smartphone only exchanges short pointers via Bluetooth-Low-Energy to large pieces of memory. Our contact tracing system relying on the use of a blockchain offers complete transparency and resilience through full decentralization, therefore being more appealing for citizens.

Chapter 7 faces a different problem still related to contact tracing systems. We show that an adversary can attack the integrity of contact tracing systems based on Google-Apple Exposure Notifications (GAEN) by leveraging blockchain technology. We show that through smart contracts there can be an on-line market where infected individuals interested in monetizing their status can upload to the servers of the GAEN-based systems some keys (i.e., Temporary Exposure Keys) chosen by a non-infected adversary. This contribution is part of a paper accepted to ACNS '21 [AFV21].

Practical Verifiable MPC using Bulletproofs In Chapter 8 we consider the setting where multiple parties have a secret input (a party that has an input is called *input party*) and want to evaluate a function over these inputs while: 1) keeping the inputs private and 2) delegating the computation to a different set of entities.

We present a practical scheme in which the task of the input party is reduced to a minimum: The input party posts a single encrypted message to a bulletin board. The compute parties then apply a threshold cryptosystem to transform the encryption to secret shares, to be used as input for the secure computation. The compute parties also produce a zero-knowledge proof of correctness of the computation that allows anyone, particularly someone external to the secure computation, to check the correctness of the output, while preserving the privacy properties of the MPC protocol. This work extends and supersedes PRIViLEDGE deliverable D3.2, Chapter 9, and D4.3, Chapter 5, by providing more details on the practical construction. Part of these results have been used for Use Case 2.

Verifiable Multi-Party Business Processes Business process automation (BPA) is the use of technology to execute recurring tasks or processes with the goal of replacing manual effort. Most BPA deployments aim to automate a firm's internal operations. However, many business processes are composed of a series of steps taken by different firms.

In Chapter 9 we propose an approach that is more scalable and can be used as the basis for many of the multiparty process enforcement use cases that others address through DLT (which create problems in terms of performances and privacy). Our solution is based on committing process states into an authenticated data structure operated by a server whose actions can be independently verified. This content of this chapter was presented at the Third Workshop on Security and Privacy-Enhanced Business Process Management of the International Conference on Business Process Management [SST20].

Chapter 2

Updatable Blockchains

Software updates for blockchain systems become a real challenge when they impact the underlying consensus mechanism. The activation of such changes might jeopardize the integrity of the blockchain by resulting in chain splits. Moreover, the software update process should be handed over to the community and this means that the blockchain should support updates without relying on a trusted party. In this chapter, we introduce the notion of *updatable blockchains* and show how to construct blockchains that satisfy this definition. Informally, an updatable blockchain is a secure blockchain and in addition it allows to update its protocol preserving the history of the chain. In this work, we focus only on the processes that allow securely switching from one blockchain protocol to another assuming that the blockchain protocols are correct. That is, we do not aim at providing a mechanism that allows reaching consensus on what is the code of the new blockchain protocol. We just assume that such a mechanism exists (like the one proposed in NDSS 2019 by Zhang et. al), and show how to securely go from the old protocol to the new one. The contribution of this chapter can be summarized as follows. We provide the first formal definition of updatable ledgers and propose the description of two compilers. These compilers take a blockchain and turn it into an updatable blockchain. The first compiler requires the structure of the current and the updated blockchain to be very similar (only the structure of the blocks can be different) but it allows for an update process that is more simple and efficient. The second compiler that we propose is very generic (i.e., makes few assumptions on the similarities between the structure of the current blockchain and the updated blockchain). The drawback of this compiler is that it requires the new blockchain to be resilient against a specific adversarial behaviour and requires all the honest parties to be online during the update process. However, we show how to get rid of the latest requirement (the honest parties being online during the update) in the case of proof-of-work and proof-of-stake ledgers.

2.1 Introduction

Most of the existing software requires to be updated (or replaced) at some point. Indeed, the most vital aspect for the sustainability of any software system is its ability to effectively and swiftly adapt to changes; one basic form of which are software updates. Therefore, the adoption of software updates is at the heart of the lifecycle of any system, and blockchain systems are no exception. Software updates might be triggered by a plethora of different reasons: change requests, bug-fixes, security holes, new-feature requests, various optimizations, code refactoring etc. More specifically, for blockchain systems, a typical source of change is the enhancements at the consensus protocol level. There might be changes to the values of specific parameters (e.g., the maximum block size, or the maximum transaction size etc.), changes to the validation rules at any level (transaction, block, or blockchain), or even changes at the consensus protocol itself. Usually, the reason for such changes is the reinforcement of the protocol against a broader scope of adversary attacks, or the optimization of some aspect of

the system like the transaction throughput, or the storage cost etc. A software update's lifecycle comprises of three important decision points: a) what update proposal should be implemented, b) is a specific implementation appropriate to be deployed and c) when and how the changes should be activated on the blockchain. A fully decentralized approach should decentralize all of these three decisions. Indeed, there are already proposals on how to update specific blockchain protocols in a decentralized way [DD18, Dec19, Goo14]. Moreover, Bingsheng et al. [ZOB19], proposes a complete treasury system in order to solve the funding problem for software updates. The decentralization of such decisions is usually called in short *decentralized governance*. This chapter does not focus on how to achieve decentralized governance for software updates. Indeed, we assume that appropriate decentralized governance processes (e.g., voting, delegation of voting, upgrade-readiness signaling etc.) are in place and the community has already reached a consensus on what specific update should be activated and this information is *written* on the blockchain. Moreover, we assume that a sufficient percent of honest parties have expressed (e.g. through a signaling mechanism) their readiness to upgrade to the new ledger. This is exactly the point from where our focus begins. In particular, we deal with the *secure activation* of software update changes on the blockchain in a fully decentralized setting and essentially provide a way to safely transition from the old ledger to the upgraded ledger without the need of a trusted third party (TTP). Moreover, we define what is a secure activation of changes by introducing the notion of *updatable blockchains*. To the best of our knowledge, our approach is the first that treats the problem of decentralized activation of updates for blockchains in such a formal way providing a security definition for updatable blockchain and generic constructions (more details will be provided in the next section).

2.1.1 Our Contributions

In our work, we try to define what is a ledger[1] that supports updates and refer to it as an *updatable ledger*.

Then we propose a generic compiler that takes a ledger $\mathcal{L}_1$ and turns it into an updatable ledger that tolerates updates only with respect to ledgers that follow the same consensus rule as $\mathcal{L}_1$ but have different block structure. We then propose another (more generic) compiler that, always starting from $\mathcal{L}_1$, turns $\mathcal{L}_1$ it into a ledger $\mathcal{L}^{\mathsf{UPD}}$ that can be updated to the code of a ledger $\mathcal{L}_2$. This compiler works assuming only few similarities between $\mathcal{L}_1$ and $\mathcal{L}_2$, but it is more complicated and decreases the throughput of the ledger during the update. All our constructions do not rely on any trusted third party (TTP).

2.1.2 Our Techniques

Our definition of updatable ledgers is quite intuitive. We require an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ to be secure under the standard definition of security (i.e., it has to enjoy consistency and liveness) but on top of this, it has to support the property of *updatability*. This property guarantees that, in the case there are enough parties that are willing to upgrade the code of $\mathcal{L}^{\mathsf{UPD}}$ to the code of a new ledger $\mathcal{L}_2$, the honest parties can securely run $\mathcal{L}_2$ and preserve the state of $\mathcal{L}^{\mathsf{UPD}}$.

Clearly, (almost) any ledger $\mathcal{L}_1$ can be turned into an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ if we can rely on a TTP. Indeed, in this case the TTP can issue a genesis block for $\mathcal{L}_2$ which incorporates the state of $\mathcal{L}_1$ (or just the hash of it), and then the parties that where running $\mathcal{L}_1$ can abandon it and start running $\mathcal{L}_2$ using the genesis block issued by the TTP.

We show how to construct an updatable ledger without relying on a TTP. The starting point for our construction is a standard ledger $\mathcal{L}_1$ that we enhance with the following mechanism. At time T_0 (when enough parties are assumed to be willing to update to $\mathcal{L}_2$) a block of $\mathcal{L}_1$ is chosen and *translated* into a genesis block for $\mathcal{L}_2$. All the parties that wanted to update can now simply run $\mathcal{L}_2$ on the chosen

[1] With slight abuse of terminology we use the words ledger and blockchain interchangeably.

genesis block. This approach clearly requires that there is an efficient way to translate a block of $\mathcal{L}_1$ into a block for $\mathcal{L}_2$, and this might limit the class of ledgers to which $\mathcal{L}^{\mathsf{UPD}}$ can be updated.

Even though the above approach seems to work, there are unfortunately many subtleties that we need to deal with. The first is that the adversary might be able to see the genesis block for $\mathcal{L}_2$ before any other honest parties do, and therefore he can take advantage on the generation of the blocks of $\mathcal{L}_2$ thus compromising the security of the system. The second issue is that the adversary might influence the choice of the genesis block. Indeed, we do not know how the consensus algorithm of $\mathcal{L}_1$ works and what is the power of the adversary in biasing the content of $\mathcal{L}_1$'s blocks. We note that this scenario (where there are many candidates blocks and the adversary can decide which block is added to the final chain) is well studied (see [GKL15b]) and many blockchain protocols allow this kind of adversarial behaviour (i.e., an adversary can create forks and influence the decision on what fork will become part of the stable chain). To tackle these issues, we further shrink the class of ledgers to which $\mathcal{L}^{\mathsf{UPD}}$ can be updated, and require $\mathcal{L}_2$ to retain its security even in the case the genesis block can be seen by the adversary before that the honest parties can see it, and even if the adversary can pick the genesis block from a set of candidate genesis blocks. Despite being quite general, this compiler has the drawback that the honest parties need to be online during the update. Indeed, if an honest party is offline before T_0 and comes online after the update then no security can be guaranteed for this party. However, we show how to relax the requirement on the honest parties being online during the update by relying on a *2-for-1 mining* approach (more details are provided in the end of Sec. 2.4.2).

The second scheme that we propose requires $\mathcal{L}^{\mathsf{UPD}}$ and $\mathcal{L}_2$ to be the same (i.e., they use the same consensus rules) but might have a different block structure. In this case, the update process is even simpler, the parties, starting from a pre-agreed block index j, start extending the state of $\mathcal{L}^{\mathsf{UPD}}$ using the rules of $\mathcal{L}_2$ even if the block in position j is not stable. That is, it might happen that different honest parties start running $\mathcal{L}_2$ using a different starting block given that the block j does not belong to the common prefix. We prove that this does not cause issues even in the case when not all the honest parties participate in the update (i.e., some honest parties are offline or decided to not participate to the update). The advantage of this approach over the first that we have proposed is that we do not require all the honest parties to be online during the update, and the throughput is not affected by the update process.

2.2 The Model

Protocol participants are represented as parties—formally Interactive Turing Machine instances (ITIs)—in a multi-party computation. We assume a central adversary who corrupts parties and uses them to attack the protocol. The adversary is *adaptive*, i.e., can corrupt (additional) parties at any point and depending on his current view of the protocol execution. Our protocols are synchronous (G)UC protocols [BMTZ17, KMTZ13]: parties have access to a (global) clock setup, denoted by $\mathcal{G}_{\mathtt{clock}}$, and can communicate over a network of authenticated multicast channels. We note that the assumption on the existence of a global clock has been used to prove the security of Bitcoin [BMTZ17] and we are not aware of any other formal proof that relies on weaker notion of "time". For this reason we believe that the use of the functionality $\mathcal{G}_{\mathtt{clock}}$ in this work is without loss of generality.

We assume instant and *fetch-based* delivery channels [KMTZ13, CGHZ16]. Such channels, whenever they receive a message from their sender, they record it and deliver it to the receiver upon his request with a "fetch" command. In fact, all functionalities we design in this work will have such fetch-based delivery of their outputs. We remark that the instant-delivery assumption is without loss of generality as the channels are only used for communicating the timestamped object to the verifier which can anyway happen at any point after its creation. However, our treatment trivially applies also to the setting where parties communicate over bounded-delay channels as in [BMTZ17].

Functionalities with Dynamic Party Sets UC provides support for functionalities in which the set of parties that might interact with the functionality is dynamic. We make this explicit by means of the following mechanism (that we describe almost verbatim from [BMTZ17, Sec. 3.1]): All the functionalities considered here include the following instructions that allow honest parties to join or leave the set $\mathcal{P}$ of players that the functionality interacts with, and inform the adversary about the current set of registered parties:

- Upon receiving $(\texttt{REGISTER}, sid)$ from some party p_i (or from $\mathcal{A}$ on behalf of a corrupted p_i), set $\mathcal{P} := \mathcal{P} \cup \{p_i\}$. Return $(\texttt{REGISTER}, sid, p_i)$ to the caller.
- Upon receiving $(\texttt{DE\_REGISTER}, sid)$ from some party $p_i \in \mathcal{P}$, the functionality updates $\mathcal{P} := \mathcal{P} \setminus \{p_i\}$ and returns $(\texttt{DE\_REGISTER}, sid, p_i)$ to p_i.
- Upon receiving $(\texttt{IS\_REGISTERED}, sid)$ from some party p_i, return $(\texttt{REGISTER}, sid, b)$ to the caller, where the bit b is 1 if and only if $p_i \in \mathcal{P}$.
- Upon receiving $(\texttt{GET\_REGISTERED}, sid)$ from $\mathcal{A}$, the functionality returns the response $(\texttt{GET\_REGISTERED}, sid, \mathcal{P})$ to $\mathcal{A}$.

In addition to the above registration instructions, global setups, i.e., shared functionalities that are available both in the real and in the ideal world and allow parties connected to them to share state [CDPW07], allow also UC functionalities to register with them. Concretely, global setups include, in addition to the above party registration instructions, two registration/de-registration instructions for functionalities:

- Upon receiving $(\texttt{REGISTER}, sid_G)$ from a functionality F (with session-id sid), update $F := F \cup \{(F, sid)\}$.
- Upon receiving $(\texttt{DE\_REGISTER}, sid_G)$ from a functionality F (with session-id sid), update $F := F\{(F, sid)\}$.
- Upon receiving $(\texttt{GET\_REGISTERED}_F, sid_G)$ from $\mathcal{A}$, return $(\texttt{GET\_REGISTERED}_F, sid_G, F)$ to $\mathcal{A}$.

We use the expression sid_G to refer to the encoding of the session identifier of global setups. By default (and if not otherwise stated), the above four (or seven in case of global setups) instructions will be part of the code of all ideal functionalities considered in this work. However, to keep the description simpler we will omit these instructions from the formal descriptions unless deviations are defined.

The Clock Functionality $\mathcal{G}_{\texttt{clock}}$ (cf. Fig. 2.2). The *clock functionality* was initially proposed in [KMTZ13] to enable synchronous execution of UC protocols. Here we adopt its global-setup version, denoted by $\mathcal{G}_{\texttt{clock}}$, which was proposed by [BMTZ17] and was used in the (G)UC proofs of the ledger's security.[2] $\mathcal{G}_{\texttt{clock}}$ allows parties (and functionalities) to ensure that the protocol they are running proceeds in synchronized rounds; it keeps track of round variable whose value can be retrieved by parties (or by functionalities) via sending to it the pair: `CLOCK-READ`. This value is increased when every honest party has sent to the clock a command `CLOCK-UPDATE`. The parties use the clock as follows. Each party starts every operation by reading the current round from $\mathcal{G}_{\texttt{clock}}$ via the command `CLOCK-READ`. Once any party has executed all its instructions for that round it instructs the clock to advance by sending a `CLOCK-UPDATE` command, and gets in an idle mode where it simply reads the clock time in every activation until the round advances. To keep more compact the description of our functionalities that rely on $\mathcal{G}_{\texttt{clock}}$, we implicitly assume that whenever an input is received the command `CLOCK-READ` is sent to $\mathcal{G}_{\texttt{clock}}$ to retrieve the current round. Moreover, before giving the output, the functionalities request to advance the clock by sending `CLOCK-UPDATE` to $\mathcal{G}_{\texttt{clock}}$.

[2] As a global setup, $\mathcal{G}_{\texttt{clock}}$ also exists in the ideal world and the ledger connects to it to keep track of rounds.

2.2.1 Ledger Consensus: Model

In this section, we define our notion of protocol execution following [GKL15b, Can01]. The execution of a protocol Π is driven by an environment program $\mathcal{Z}$ that may spawn multiple instances running the protocol Π. The programs in question can be thought of as interactive Turing machines (ITM) that have communication, input and output tapes. An instance of an ITM running a certain program will be referred to as an interactive Turing machine instance or ITI. The spawning of new ITI's by an existing ITI as well as the interaction between them is at the discretion of a control program which is also an ITM and is denoted by C. The pair $(\mathcal{Z}, C)$ is called a system of ITM's, cf. [Can01]. Specifically, the execution driven by $\mathcal{Z}$ is defined with respect to a protocol Π, an adversary $\mathcal{A}$ (also an ITM) and a set of parties $P_1, \ldots, P_n$; these are hardcoded in the control program C. Initially, the environment $\mathcal{Z}$ is restricted by C to spawn the adversary $\mathcal{A}$. Each time the adversary is activated, it may send one or more messages of the form $(\mathsf{corrupt}, P_i)$ to C. The control program C will register party P_i as corrupted, only provided that the environment has previously given an input of the form $(\mathsf{corrupt}, P_i)$ to $\mathcal{A}$ and that the number of corrupted parties is less or equal tc, a bound that is also hardcoded in C.

We divide time into discrete units called *time slots* or *r*ound. Players are equipped with (roughly) synchronized clocks $\mathcal{G}_{\mathtt{clock}}$ that indicate the current slot: we assume that any clock drift is subsumed in the slot length.

Ledger Consensus. Ledger consensus (a.k.a. "Nakamoto consensus") is the problem where a set of nodes (or parties) operate continuously accepting inputs that are called transactions and incorporate them in a public data structure called the *ledger*. A ledger (denoted in calligraphic-face, e.g. $\mathcal{L}$) is a mechanism for maintaining a sequence of transactions, often stored in the form of a blockchain. In this work, we denote with $\mathcal{L}$ the algorithms used to maintain the sequence, and with L all the views of the participants of the state of these algorithms when being executed. For example, the (existing) ledger Bitcoin consists of the set of all transactions that ever took place in the Bitcoin network, the current UTXO set, as well as the local views of all the participants. In contrast, we call a *ledger state* a concrete sequence of transactions $\mathsf{Tx}_1, \mathsf{Tx}_2, \ldots$ stored in the stable part of a ledger state L, typically as viewed by a particular party. Hence, in every blockchain-based ledger $\mathcal{L}$, every fixed chain $\mathcal{C}$ defines a concrete ledger state by applying the interpretation rules given as a part of the description of $\mathcal{L}$. In this work, we assume that the ledger state is obtained from the blockchain by dropping the last k blocks and serializing the transactions in the remaining blocks. We refer to k as the *common-prefix parameter*. We denote by $\mathsf{L}^P[t]$ the ledger state of a ledger $\mathcal{L}$ as viewed by a party P at the beginning of a time slot t and by $\check{\mathsf{L}}^P[t]$ the complete state of the ledger (at time t) including all pending transactions that are not stable yet. $\mathsf{L}^P[t]$ can be obtained from $\check{\mathsf{L}}^P[t]$ by dropping the last k block.

For two ledger states (or, more generally, any sequences), we denote by $\preceq$ the prefix relation. Recall the definition of secure ledger protocol given in [GK20].

Definition 1. *A ledger protocol $\mathcal{L}$ is secure if it enjoys the following properties.*

> **Consistency.** *For any two honest parties P_1, P_2 and two time slots $t_1 \leq t_2$, it holds $\mathsf{L}^{P_1}[t_1] \preceq \check{\mathsf{L}}^{P_2}[t_2]$.*
>
> **Liveness.** *If all honest parties in the system attempt to include a transaction Tx then, at any slot t after s slots (called the liveness parameter), any honest party P, if queried, will report $\mathsf{Tx} \in \mathsf{L}^P[t]$.*

In this work we also explicitly rely on the properties of *Common Prefix (CP)*, *Chain Growth (CG)* and *Chain Quality (CQ)*.

> **Common Prefix (CP); with parameters** $k \in \mathbb{N}$ states that for any pair of honest players P_1, P_2 at rounds $r_1 \leq r_2$ respectively, it holds that $\mathsf{L}^{P_1}[r_1] \preceq \check{\mathsf{L}}^{P_2}[r_2]$.

Chain Growth (CG); with parameters $\tau \in (0,1]$ **and** $s \in \mathbb{N}$. Consider the chain $\mathcal{C}$ adopted by an honest party at the onset of a slot and any portion of $\mathcal{C}$ spanning s prior slots; then the number of blocks appearing in this portion of the chain is at least τs.

Chain Quality (CQ) with parameters $\mu \in \mathbb{R}$ **and** $\ell \in \mathbb{N}$. For any honest party P with chain $\mathcal{C}$ it holds that for any ℓ consecutive blocks of $\mathcal{C}$ the ratio of honest blocks is at least μ.

We consider a setting where a set of parties run a protocol maintaining a ledger $\mathcal{L}_1$. Following [GKZ19], we denote by $\mathbb{A}_1$ the assumption for $\mathcal{L}_1$. That is, if the assumption $\mathbb{A}_1$ holds, then ledger $\mathcal{L}_1$ is secure under the Definition 1. Formally, $\mathbb{A}_i$ for a ledger $\mathcal{L}_i$ is a sequence of events $\mathbb{A}_i[t]$ for each time slot t that can assume value 1, if the assumption is satisfied, and 0 otherwise. For example, $\mathbb{A}_i$ may denote that there has never been a majority of hashing power (or stake in a particular asset, on this ledger or elsewhere) under the control of the adversary; that a particular entity (in case of a centralized ledger) was not corrupted; and so on. Without loss of generality, we say that the assumption $\mathbb{A}_1$ for the ledger $\mathcal{L}_1$ holds if and only if the fraction of corrupted parties (the parties that received the input $(\mathsf{corrupt}, \cdot)$) is below the threshold tc_1 (where tc_1 is part of the control function as described in the beginning of this section).

Chain selection rule and block validation. We sometimes assume that a ledger protocol describes a *chain selection rule* that we denote with ChainSel. That is, we assume that each party in each round of the execution of the protocol collects all chains that come from the network and runs the algorithm ChainSel to decide whether to keep his current local chain $\mathsf{C}_{\mathsf{loc}}$, or adopt one of the newly received chains. Following [BMTZ17] we also assume that before applying the chain-selection rule, any given chain is tested using the procedure IsValidChain. IsValidChain checks filters the valid chains among all the chains received from the network and only the valid chain are used as input for ChainSel. ChainSel in turns rely on the algorithm IsValidBlock. IsValidBlock take as input a block B of $\mathsf{C}_{\mathsf{loc}}$ and outputs 1 if B is a valid block (i.e., the structure of the block is correct) and 0 otherwise.

We note that by assuming that a ledger protocol is always equipped with the algorithms ChainSel, IsValidChain and IsValidBlock make some of our results less general. However, we will show that it is possible to obtain a better updatable ledger in the case when the two ledgers (the current ledger) and the new ledger have the same chain selection rule (among other similarities).

2.2.2 Genesis Block Functionality

The ledger protocols that we consider in this work are equipped with the description of an algorithm gen_genesis that, on input a random value of appropriate length, outputs a valid genesis block (i.e., the first block of the chain). The security of most of the known ledger protocols holds under the additional assumption that the genesis block is correct. That is, the genesis block has been generated accordingly to gen_genesis using appropriate randomness. Multiple ways have been presented to generate a correct genesis block in the literature (i.e., by relying on a trusted authority, use unpredictable information (like in bitcoin), run a multi-party computation (MPC) protocol [zca], rely on PoW [GKLP18] assumptions and so on and so forth). In this work we abstract the generation of the genesis block by means of an ideal functionality. The ideal functionality that one might expect, upon being activated from the adversary or from an honest party, should sample a random string and use it to run the algorithm gen_genesis. Unfortunately this simple functionality does not cover real world scenarios where an adversarial party might see the genesis block before the honest parties do. This, for example, can happen in the case when gen_genesis is realized via an MPC protocol and a rushing adversary[3] could hold the genesis

[3] A rushing adversary waits to receive the messages from all the honest parties and then computes its reply. Note that this means that, in general, the adversary is always able to see the output of the computation before the honest parties do.

Genesis Functionality for $\mathcal{L}$
Parameters. The functionality is parametrized by τ^{max}, the maximum number of candidate genesis block m, the genesis block B^{gen} initialized with a default value $\perp$ and the procedure gen_genesis(). We assume the functionality to be registered to $\mathcal{G}_{\mathtt{clock}}$ and that it maintains a set of registered parties $\mathcal{P}$. On any input I the functionality queries $\mathcal{G}_{\mathtt{clock}}$, and we denote with R be the response obtained by $\mathcal{G}_{\mathtt{clock}}$.

- If $I = \mathtt{GEN\_GENESIS}$ is received from the adversary $\mathcal{A}$ then set $\tau := R$, generate m genesis blocks (each block is generated by running the procedure gen_genesis()) $\mathsf{GB} := \{B_1^{\mathsf{gen}}, \ldots, B_m^{\mathsf{gen}}\}$ for $\mathcal{L}$, and send GB to the adversary.
- If $I = \mathtt{GET\_GENESIS}$ is received from an honest party $p_i \in \mathcal{P}$ do the following
  - If $B^{\mathsf{gen}} \neq \perp$ then return B^{gen} to p_i.
  - If $B^{\mathsf{gen}} = \perp$ and $R - \tau > \tau^{\mathsf{max}}$ then generate a genesis block $\tilde{B^{\mathsf{gen}}}$ by running gen_genesis(), set $B^{\mathsf{gen}} \leftarrow \tilde{B^{\mathsf{gen}}}$ and send B^{gen} to p_i.
- If $I = (\mathtt{SET\_GENESIS}, B^{\mathsf{gen}\prime})$ is received from the adversary do the following
  - If $(R - \tau) \leq \tau^{\mathsf{max}}$ and $B^{\mathsf{gen}\prime} \in \mathsf{GB}$ then set $B^{\mathsf{gen}} := B^{\mathsf{gen}\prime}$.
  - Else, return $\perp$ to the adversary.

Figure 2.1: The genesis functionality $\mathcal{F}^{\mathsf{gen}}$.

block (the output of the computation) for some bounded amount of time τ^{max} before the honest parties can see it. We note that an adversary can use this strategy to take an advantage on the generation of the blocks that extend the genesis block. Therefore, the first modification that we consider for our ideal functionality is to allow the adversary to see the genesis block up to τ^{max} rounds earlier than the honest parties. The second relaxation allows the adversary to see up to m honestly generated genesis blocks and consequently decide which of these blocks will become the genesis block. We propose the formal description of our genesis functionality $\mathcal{F}^{\mathsf{gen}}$ in Fig. 2.1. We note that the case where $\tau^{\mathsf{max}} = 0$ and $m = 1$ corresponds to the case where there is only one candidate genesis block and all the parties can see it at the same round.

2.3 Secure Updatable Ledgers

2.3.1 Defining Secure Updatable Ledgers

In this section, we provide the definition of updatable ledgers. Our definition is generic in the sense that can be applied to a large class of ledgers (e.g., PoS, PoW and so on). Let $\mathcal{L}^{\mathsf{UPD}}$ and $\mathcal{L}_2$ be the two ledgers with the respective assumptions $\mathbb{A}_1$ and $\mathbb{A}_2$. Assuming that $\mathbb{A}_1$ holds, then among the parties that are running $\mathcal{L}^{\mathsf{UPD}}$ we could have up to a fraction of tc_1 corrupted parties (i.e., parties that have received the command corrupt). Analogously, the assumption $\mathbb{A}_2$ for the ledger $\mathcal{L}_2$ holds if the number of corrupted parties divided by the number of honest parties is below the threshold tc_2.

The interface of an updatable ledger extends the interface of a standard ledger by adding the command $(\mathsf{activate}, \mathcal{L}_2)$. That is, each party that runs an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ can receive the command $(\mathsf{activate}, \mathcal{L}_2)$ from the environment to enable the update procedure. Let t_{P_i} denote the time in which a party P_i receives the activation command and let $\mathcal{P}^{\mathsf{u}}$ be the set of parties that received this command. Informally, an updatable ledger guarantees that if the set of honest parties that are willing

The functionality is available to all participants. The functionality is parametrized with variable τ, a set of parties $\mathcal{P} = p_1, \ldots, p_n$, and a set F of functionalities. For each party $p_i \in \mathcal{P}$ it manages variable d_i. For each $\mathcal{F} \in F$ it manages variable $d_{\mathcal{F}}$
Initially, $\tau = 0, \mathcal{P} = \emptyset$ and $F = \emptyset$.

- Upon receiving $(\texttt{CLOCK-UPDATE}, sid)$ from some party $p_i \in \mathcal{P}$ set $d_i = 1$ execute *Round-Update* and forward $(\texttt{CLOCK-UPDATE}, sid, p_i)$ to $\mathcal{A}$.
- Upon receiving $(\texttt{CLOCK-UPDATE}, sid)$ from some functionality $\{ \in F$ set $d_{\mathcal{F}} = 1$, execute *Round-Update* and return $(\texttt{CLOCK-UPDATE}, sid, F)$ to F.
- Upon receiving $(\texttt{CLOCK-READ}, sid)$ from any participant (including the environment, the adversary, or any ideal-shared or local-functionality) return $(\texttt{CLOCK-READ}, sid, \tau)$ to the requester.

Procedure *Round-Update*: If $d_{\mathcal{F}} = 1$ for all $\mathcal{F} \in F$ and $d_i = 1$ for all honest $p_i \in \mathcal{P}$, then set $\tau = \tau + 1$ and reset $d_{\mathcal{F}} = 0$ and $d_i = 0$ for all parties in $\mathcal{P}$.

Figure 2.2: The functionality $\mathcal{G}_{\texttt{clock}}$

to run $\mathcal{L}_2$ (i.e., the number of parties that received $(\mathsf{activate}, \mathcal{L}_2)$) is such that $\mathbb{A}_2[\tau] = 1$ for all $\tau \geq T_0$ for some $T_0 \in \mathbb{N}$, then the state of $\mathcal{L}_2$ at time $T_0 + \Delta$ corresponds to the state of $\mathcal{L}^{\mathsf{UPD}}$ at some time $T \in [T_0, T_0 + \Delta]$. The parameter Δ represents the time required for the update process to be completed. The above implies that L_2 extends L_1 and that $\mathcal{L}_2$ is secure (i.e., it enjoys consistency and liveness). In a nutshell, a secure update process guarantees that the state of the old ledger is moved into the new ledger, and that the new ledger is secure. We now give a more formal definition.

Definition 2 (Updatable Ledger)**.** *We say that a ledger $\mathcal{L}^{\mathsf{UPD}}$ is updatable with activation parameter Δ (where $\Delta \in \mathbb{N}$) if it is a secure ledger according to Def. 1 and it enjoys the following property.*

Updatability. *Let $\mathcal{L}_2$ be a secure ledger (always according to Def. 1). Let $\mathcal{P}^{\mathsf{u}}$ be the set of parties that received the input $(\mathsf{activate}, \mathcal{L}_2)$. If $\mathcal{P}^{\mathsf{u}}$ is such that $\mathbb{A}_2[\tau] = 1$ for all $\tau \geq T_0$ for some $T_0 \in \mathbb{N}$ and $\mathbb{A}_1[\tau'] = 1$ for all $\tau' \leq T_1 = T_0 + \Delta$, then*

1. *$\mathsf{L}_1^{P_i}[T'] \preceq \mathsf{L}_2$ for some $P_i \in \mathcal{P}^{\mathsf{u}}$ with $T_0 \leq T' \leq T_1$.*
2. *for all $\tau'' \geq T_1$ $\mathcal{L}_2$ enjoys consistency and liveness*

We note that this definition says nothing on the security of $\mathcal{L}^{\mathsf{UPD}}$ after the time $T_1 = T_0 + \Delta$. Indeed, the Definition 2 implies that if after this time slot $T_0 + \Delta$ $\mathcal{L}^{\mathsf{UPD}}$ becomes insecure (e.g., because $\mathbb{A}_1$ does not hold) then the security of $\mathcal{L}_2$ is not compromised.

We relax the above definition by introducing the notion of updatable ledger in the *semi-online* setting. An updatable ledger in the semi-online setting guarantees the properties of updatability only for the honest parties that where active during the activation period $[T_0, T_1]$. That is, if an honest party P is offline before time T_0, and comes online after at time T_1 then no security is guaranteed with respect to P.

2.4 Our Constructions

In this section we propose two main approaches to turn a ledger $\mathcal{L}_1$ into an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$. That is, we show how to make $\mathcal{L}_1$ able to self-update to the code of a new ledger $\mathcal{L}_2$. The first approach

proposed requires $\mathcal{L}_1$ and $\mathcal{L}_2$ to be the same (i.e., they use the same consensus rules) but might have a different block structure. The advantage in this approach is that we get a very simple updatable ledger, that does not decrease the throughput of $\mathcal{L}^{\mathsf{UPD}}$ during the update and does not require all the honest parties to be online during the update[4]. The second approach requires fewer similarities between the two ledgers, but it is proven secure only in the semi-online setting. We also show that we can relax the requirement on the honest parties being online during the update by relying on a *2-for-1 mining* approach (more details are provided in the end of Sec. 2.4.2).

We now provide a detailed description of our approaches and formally prove their security.

2.4.1 First Approach

In this subsection we consider a simplified scenario where the two ledgers, $\mathcal{L}_1$ and $\mathcal{L}_2$, are the same except for the block format (i.e., $\mathcal{L}_1$ and $\mathcal{L}_2$ might have a different block size). Moreover, we assume that a block valid for $\mathcal{L}_1$ is valid for $\mathcal{L}_2$ as well (but the vice versa does not necessarily hold). Formally, this means that if the block validation algorithm $\mathsf{IsValidBlock}_1$ of $\mathcal{L}_1$ outputs 1 on some input B, then also the block validation algorithm $\mathsf{IsValidBlock}_2$ of $\mathcal{L}_2$ outputs 1 (see Sec. 2.2.1 for more details). We now prove the following theorem

Theorem 1. *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are secure ledgers with block validation rules respectively $\mathsf{IsValidBlock}_1$ and $\mathsf{IsValidBlock}_2$ such that:*

1. *$\mathcal{L}_1$ and $\mathcal{L}_2$ are the same except with respect to the block validation rules;*
2. *for every block B such that if $\mathsf{IsValidBlock}_1(B) = 1$ then $\mathsf{IsValidBlock}_2(B) = 1$,*
3. *$\mathcal{L}_1$ (resp. $\mathcal{L}_2$) has common-prefix parameter k, chain-growth parameter (τ, s) and assumption $\mathbb{A}_1$ (resp. $\mathbb{A}_2$) with $\mathbb{A}_1 = \mathbb{A}_2$,*

then there exists an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ with update parameter $\Delta := (k+1)\tau^{-1} + s$.

Proof. We assume that enough parties have received the command $(\mathsf{activate}, \mathcal{L}_2)$ such that $\mathbb{A}_2$ holds and denote the time when this happen with T_0. Our updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ works as follows.

Each party $P_i \in \mathcal{P}^{\mathsf{u}}$ does the following steps.

1. Use $\mathsf{IsValidBlock}_2$ as a block validation algorithm.
2. Create and post a transaction that contains an *activation flag*.
3. Let $\mathsf{i_f}$ be the index of the block that will contain the first transaction with an activation flag.
4. Let $j := \mathsf{i_f} + k + 1$, run $\mathcal{L}_1$ and when the j-th block B_j^i becomes part of $\check{\mathsf{L}}_1^{P_i}[\tau_i]$ for some $\tau_i \geq T_0$ start extending B_j^i using the rules of $\mathcal{L}_2$ instead of the rules of $\mathcal{L}_1$ (we recall that a valid block for $\mathcal{L}_1$ is also a valid block for $\mathcal{L}_2$)

We provide a pictorial description of what happens to the ledger state during the update in Fig. 2.3. We note that two honest parties P_1 and P_2 might have different $\check{\mathsf{L}}_1^{P_1}[\tau]$ and $\check{\mathsf{L}}_1^{P_2}[\tau]$ at any time τ. The Fig. 2.3 describe the scenario where P_1 might start to run $\mathcal{L}_2$ starting from an unstable block (i.e. a block of $\check{\mathsf{L}}_1^{P_1}[\tau]$ with $\tau \geq T_0 + s$) which is different from the block that P_2 is using. However, after sufficiently many rounds (at some round $\tau' \leq T_0 + s + (k_1+1)\tau_1^{-1}$ to be precise) P_1 and P_2 will agree on what is the last block of $\mathcal{L}_1$ and what is the first bock of $\mathcal{L}_2$.

[4]We also show that we can relax the requirement on the honest parties being online during the update for the case of PoW ledgers.

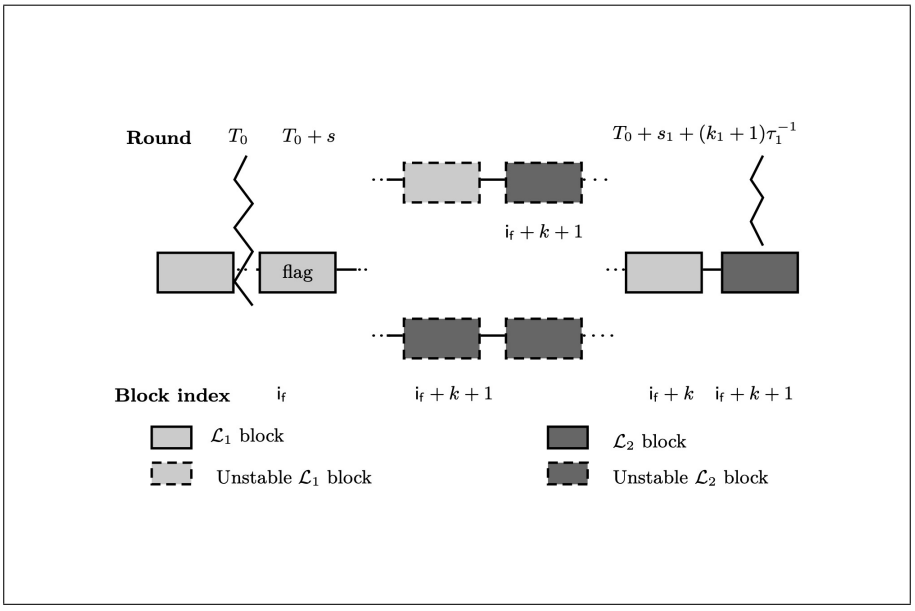


Figure 2.3: Transition from $\mathcal{L}_1$ to $\mathcal{L}_2$. Note that different honest parties might have different views (i.e., forks) of the unstable part of the chain which have also different lengths.

To complete the proof we need to show that $\mathcal{L}_2$ enjoys consistency and liveness and that the state $\mathcal{L}_1$ at some time $\tau \in [T_0, T_1]$ is a prefix of $\mathcal{L}_2$'s state.

Before doing that, we introduce the notion of *canonical execution* for the ledger $\mathcal{L}_2$. A canonical execution represents a standalone execution of $\mathcal{L}_2$. More precisely, we assume the existence of a genesis block for $\mathcal{L}_2$ (that the adversary and the honest party see at the round 0) and that $\mathbb{A}_2[\tau]$=1 for all $\tau \geq 0$. Let $\mathcal{P}$ be the set of parties that is running $\mathcal{L}_2$. Also, let t be the smallest time slot in which $B_{\mathsf{i_f}}$ appears in $\mathsf{L}_2^{P_i}[t]$ for all $P_i \in \mathcal{P}$ and let $\tilde{t}_{i,j}$ be the smallest time slot in which B_j^i appears in $\check{\mathsf{L}}_2^{P_i}[t_{i,j}]$ for each $P_i \in \mathcal{P}$ with $j := \mathsf{i_f} + k + 1$.

We now go back to our updatable ledger protocol. In the protocol that we have described, by assumption, we have that $\mathbb{A}_2[T_0] = 1$ for all $\tau \geq T_0$. From the moment when $\mathbb{A}_2$ becomes true the activation process takes $\Delta \leq (k+1)\tau^{-1} + s$ time slots to be completed.

This is because the parties need to wait for the block $\mathsf{i_f}$ to be part of all the honest parties stable view and wait for the j-th block (with $j := \mathsf{i_f} + k + 1$) of to be part of $\check{\mathsf{L}}_1^{P_i}[t_{i,j}]$ for all P_i with $t_{i,j} \in \mathbb{N}$. Note that in the moment that the block B_j^i becomes available to an honest party $P_i \in \mathcal{P}^{\mathsf{u}}$ (i.e., B_j^i is part of $\check{\mathsf{L}}_1^{P_i}$) then the party starts running $\mathcal{L}_2$ to extend B_j^i as described earlier (we recall that at this time slot the assumption $\mathbb{A}_2$ holds). Let $t'_{i,j}$ be the smallest time slot in which B_j^i appears in $\check{\mathsf{L}}_2^{P_i}[t'_{i,j}]$ for each $P_i \in \mathcal{P}$ with $t'_{i,j} \in \mathbb{N}$. If we consider the execution of the protocol from time T_0 and $T_0 + \Delta$ this can be seen as a canonical execution of $\mathcal{L}_2$ given that $\mathcal{L}_1$ and $\mathcal{L}_2$ follow the same rules and the same assumption, and given that $\check{\mathsf{L}}_1^{P_i}$ (and $\check{\mathsf{L}}_2^{P_i}$) contains at most k blocks more than $\mathsf{L}_1^{P_i}$ (and $\mathsf{L}_2^{P_i}$) for all $P_i \in \mathcal{P}^{\mathsf{u}}$. Hence, any advantage that the adversary has on our updatable ledger can be translated into an advantage for an adversary that is attacking $\mathcal{L}_2$, which is assumed to be secure. Note that it is crucial that the assumption that underlines the two ledger is the same. Indeed, we note that the number of honest parties that received $(\mathsf{activate}, \mathcal{L}_2)$ might be lower than the overall number of honest parties. Hence, the honest parties that are running the update procedure are less than the parties that are running $\mathcal{L}_1$ (this might happen as we do not require all the honest parties to update). However, given that $\mathbb{A}_1 = \mathbb{A}_2$, we can see the honest parties that did not receive the command $(\mathsf{activate}, \mathcal{L}_2)$ as parties controlled by the adversary as they are not following the update procedure. Luckily, this does not cause problems as even if we consider these parties as adversarial, $\mathbb{A}_1$ would still hold (given that

$\mathbb{A}_1 = \mathbb{A}_2$). Hence, we can claim that in the worst case everything that can be done by the adversary during the update can be done also in the canonical execution given that the number of honest parties in the canonical execution is the same as the number of honest parties that are performing the update.

We remark that the only difference between this and the canonical execution described above is that the blocks $B_{\mathsf{i_f}}, \ldots, B_{j-2}, B_{j-1}$ are generated using $\mathcal{L}_1$, but this does not represent an issue since we are assuming that any block of $\mathcal{L}_1$ is valid for $\mathcal{L}_2$.

□

We finally note that this protocol does not put any restriction on whether an honest party needs to be online or not during an update given that $\mathcal{L}_1$ and $\mathcal{L}_2$ have the same chain selection rule (only the block selection rule is different). One practical advantage of our approach is that if $\mathcal{L}_1$ (and $\mathcal{L}_2$) allows bootstrapping from the genesis block (like in [BGK+18]) so does our updatable ledger.

2.4.2 Second Approach

Before providing our construction we introduce the notion of *genesis-compatible* ledgers. We say that two ledgers $\mathcal{L}_1$ and $\mathcal{L}_2$ are genesis-compatible if a block of $\mathcal{L}_1$ can be turned into a valid candidate genesis block for $\mathcal{L}_2$. We now propose a formal definition.

Definition 3. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two secure ledgers where $\mathcal{F}^{\mathsf{gen}}$ is the genesis functionality of $\mathcal{L}_2$ parameterized by the algorithm* gen_genesis() *(see Fig. 2.1).*

We say that $\mathcal{L}_1$ is genesis-compatible with $\mathcal{L}_2$ if there exists a deterministic polynomial time algorithm $\Pi^{1\to 2}$ that, on input a valid block B of $\mathcal{L}_1$ outputs a valid genesis block $\tilde{B}$ for $\mathcal{L}_2$. Moreover, the output of $\Pi^{1\to 2}$ is identically distributed to the output of the procedure gen_genesis().

We note that $\Pi^{1\to 2}$ could be a very simple protocol. For example, if we consider two PoW ledgers that use the same puzzles, then $\mathcal{L}_1$ is genesis-compatible with $\mathcal{L}_2$ since the $\Pi^{1\to 2}$ can simply take a block of $\mathcal{L}_1$ and use it as a candidate genesis block for $\mathcal{L}_2$. We note that the definition of genesis-compatibility only tells that it is possible to generate a genesis block for $\mathcal{L}_2$ with a valid structure. That is, it does not imply that $\mathcal{L}_2$ can be securely run using any genesis block generated using $\Pi^{1\to 2}$ as, for example, using an old block of $\mathcal{L}_1$ could give an advantage to the adversary over the honest parties. More details follow.

We now propose our first compiler that turns a ledger $\mathcal{L}_1$ that is genesis-compatible with $\mathcal{L}_2$, into an updatable ledger. At a very high level our approach is the following. We use $\mathcal{L}_1$ to realize the genesis functionality of $\mathcal{L}_2$, and then we use the output of the genesis functionality to execute $\mathcal{L}_2$. We note that it is easy to create a candidate genesis block from $\mathcal{L}_1$ because it is genesis-compatible with $\mathcal{L}_2$. To complete the description of our compiler, we need to specify what block of $\mathcal{L}_1$ will be chosen, and argue that this process is indeed sufficient to realize the genesis functionality for $\mathcal{L}_2$. In our approach the parties that are running $\mathcal{L}_1$ agree on the index j of a block that will be used as a genesis block (this block can be decided using the consensus algorithm of $\mathcal{L}_1$, more details will be provided). When the block of position j, that we denote with B_j, becomes stable for all the honest parties that decided to update, then these parties use $\Pi^{1\to 2}$ to turn B_j into a genesis block for $\mathcal{L}_2$ thus obtaining B^{gen}. At this point B^{gen} is used to run $\mathcal{L}_2$ and $\mathcal{L}_1$ can be abandoned. Even though the above approach seems to work, there are many subtleties. The first is that the adversary might be able to see the block B_j before any other honest parties do, and therefore he can take an advantage on the generation of the blocks of $\mathcal{L}_2$. The second issue is that the adversary might influence the choice of the block that will appear in position j. Indeed, we do not know how the consensus algorithm of $\mathcal{L}_1$ works and what is the power of the adversary in biasing the content of B_j. We denote with $\tau^{\mathsf{max}\prime}$ the upper bound on the number of rounds that pass between the time at which the adversary can see a candidate block for $\mathcal{L}_1$ for a position j, and the time at which all the honest parties see B_j as part of the stable chain. We refer to this parameter $\tau^{\mathsf{max}\prime}$ as the *prediction* parameter. We also denote with m' the upper bound on

the number of valid chains that are broadcasted on the network that contain a block in position j and refer to this parameter as *maximum forks* parameter.

Coming back to our protocol, we note that if the genesis functionality of $\mathcal{L}_2$ is parameterized with $\tau^{\mathsf{max}} = \tau^{\mathsf{max}\prime}$ and $m = m'$ then we can prove that the solution we proposed works.

We are now ready to state formally our theorem and prove it.

Theorem 2. *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are secure ledgers and:*

1. *$\mathcal{L}_1$ has common-prefix parameter k_1, chain-growth parameter (τ_1, s_1) and assumption $\mathbb{A}_1$;*
2. *$\mathcal{L}_2$ has common-prefix parameter k_2, chain-growth parameter (τ_2, s_2) and assumption $\mathbb{A}_2$;*
3. *the prediction parameter of $\mathcal{L}_1$ is $\tau^{\mathsf{max}\prime}$ and the maximum forks parameter is m';*
4. *the genesis functionality $\mathcal{F}^{\mathsf{gen}}$ of $\mathcal{L}_2$ is parametrized by $\tau^{\mathsf{max}} = \tau^{\mathsf{max}\prime}$ and $m = m'$;*
5. *$\mathcal{L}_1$ is genesis-compatible with $\mathcal{L}_2$.*

then there exists an updatable ledger $\mathcal{L}^{\mathsf{UPD}}$ with update parameter $\Delta := 2k_1\tau_1^{-1} + s_1$ in the semi-online setting.

Proof. We start the proof by describing how formally our protocol works. Let T_0 be such that $\mathbb{A}_2$ holds. At time T_0 each party in $P_i \in \mathcal{P}^{\mathsf{u}}$ does the following steps.

1. Create and post a transaction that contains an *activation flag*, let $\mathsf{i_f}$ be the index of the block that will contain the first transaction with an activation flag (note that there might be more than one of such a transactions).
2. Keep running $\mathcal{L}_1$ until the block with index $j = \mathsf{i_f} + k_1$ becomes stable (i.e., becomes part of $\mathsf{L}_1^P[\tau]$ for all $P \in \mathcal{P}^{\mathsf{u}}$ for some $\tau \geq T_0$) and stop issuing transaction for $\mathcal{L}_1$ (if any).
3. When the j-th block B_j becomes stable then stop running $\mathcal{L}_1$ and start running $\mathcal{L}_2$ using $B^{\mathsf{gen}} \leftarrow \Pi^{1\rightarrow 2}(B_j)$ as the genesis block.

We provide a pictorial description of what happens to the ledger state during the update in Fig. 2.4. The activation flag is used by the honest parties to reach an agreement on what it will be the index of the block used as a genesis block. We note that the blocks of $\mathcal{L}_1$ that extend B_j might be unstable, moreover after the update has been completed the parties in $\mathcal{P}^{\mathsf{u}}$ will ignore the blocks of $\mathcal{L}_1$ that extend B_j (since after the update all the parties in $\mathcal{P}^{\mathsf{u}}$ will be using the rules $\mathcal{L}_2$, hence its chain selection rule). The reason why the parties in $\mathcal{P}^{\mathsf{u}}$ will stop issuing transactions for $\mathcal{L}_1$ is that these transactions might be included in blocks that extend B_j, which will be ignored after $T_0 + \Delta$ rounds. This clearly affects the throughput of the ledger in the interval $[T_0 + k_1\tau_1^{-1} + s_1, T_0 + 2k_1\tau_1^{-1} + s_1]$ (Fig. 2.4). We now continue with the proof. Let T_0 be the time at which we know that $\mathcal{P}^{\mathsf{u}}$ is such that $\mathbb{A}_2$ holds. In the worst case, the time required for an honest party to post a transaction that contains the activation flag takes time s_1 rounds (s_1 comes from the liveness of $\mathcal{L}_1$). The number of rounds required for j to be stable in the view of all the honest parties is $2k_1\tau_1^{-1}$ rounds. This is because to generate the block B_j are required at least $k_1\tau_1^{-1}$ rounds, and B_j has to be extended with at least k_1 blocks to be part of all the honest parties view (and this takes additional $k_1\tau_1^{-1}$ rounds). Hence, the time required to complete the update is $\Delta = 2k_1\tau_1^{-1} + s_1$. Once the block B_j becomes stable, the parties in $\mathcal{P}^{\mathsf{u}}$ can start running $\mathcal{L}_2$, and we are guaranteed that $\mathcal{L}_2$ enjoys liveness and consistency because the genesis block for $\mathcal{L}_2$ is created accordingly to $\mathcal{F}^{\mathsf{gen}}$ and by assumption $\mathbb{A}_2$ holds. Therefore, everything that appears before B^{gen} is preserved due to the consistency of $\mathcal{L}_2$. We refer to the state of $\mathcal{L}_1$ before B^{gen} as $\tilde{\mathsf{L}}_1$, and to the state of the ledger after the update as $\tilde{\mathsf{L}}_1||\mathsf{L}_2$. We finally note that we guarantee no security for the honest parties that were not online during the update. The reason is that after T_1 the

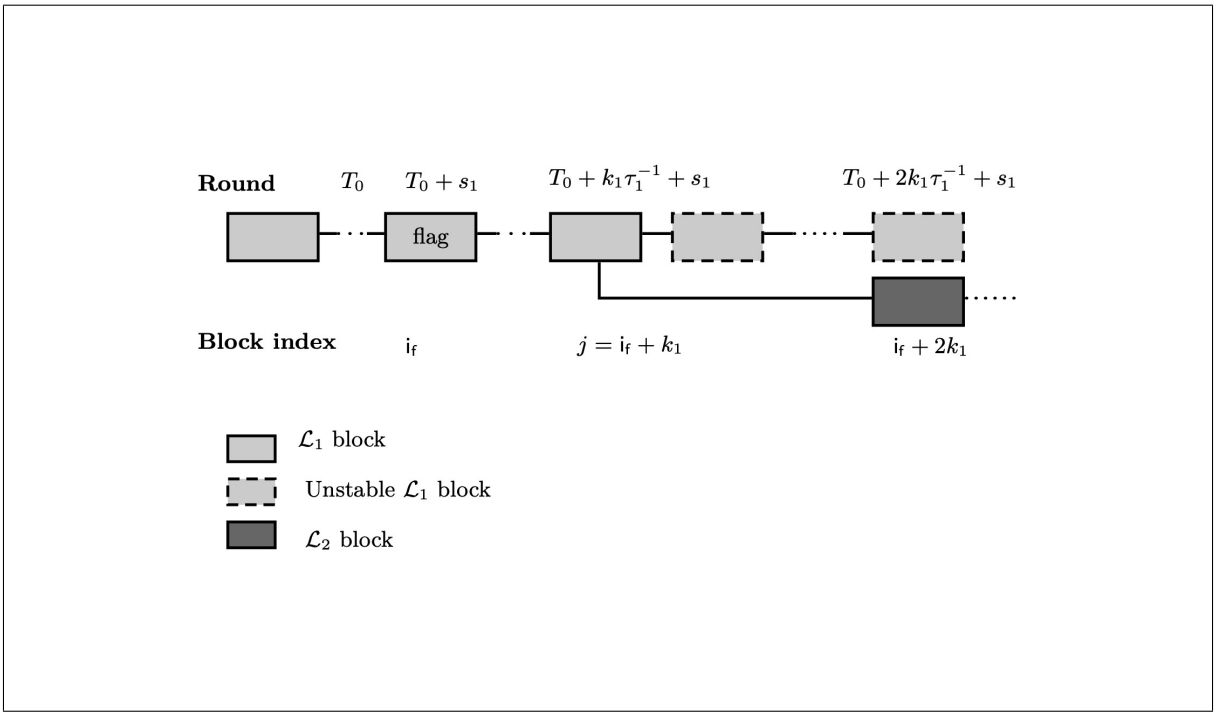


Figure 2.4: Transition from $\mathcal{L}_1$ to $\mathcal{L}_2$. Note that the empty blocks of $\mathcal{L}_1$ might be non-stable.

honest parties abandon $\mathcal{L}_1$ and the adversary could compromise it. For example, an adversary could potentially keep extending L_1 after the block j, and create a very long chain, even longer that $\tilde{\mathsf{L}}_1||\mathsf{L}_2$. Hence, if the chain selection rule of $\mathcal{L}_1$ prescribes to take the longest chain, then a party that comes online at time T_1 might take the chain L_1 (which is compromised).

□

We remark that our construction requires the parties to generate empty blocks for $\mathcal{L}_1$ from block index $j+1$ and until block B_j becomes stable. This is required as the honest parties, after the update completes, will ignore any block generated using the rules of $\mathcal{L}_1$ that comes after B_j.

Practical implications. The updatable ledger that we have described can be updated to any ledger $\mathcal{L}_2$ under the condition that the genesis functionality of $\mathcal{L}_2$ tolerates an adversary that can see the genesis block τ^{max} rounds before the honest parties and decide the genesis block among a set of m candidate genesis blocks. This requirement might look strong, but we note that the problem of constructing a ledger that is secure in such a scenario is simpler than the problem of constructing a ledger that supports temporary dishonest majority [AKWW19]. A ledger with security assumption $\mathbb{A}$ that tolerates temporary dishonest majority is such that its security properties (liveness and consistency) become valid again when $\mathbb{A}[\tau_1] = 1$, even if $\mathbb{A}[\tau'] = 0$ for all $\tau' \in [\tau_0, \tau_1 - \delta]$ for some $\tau_0, \tau_1, \delta \in \mathbb{N}$ such that $\tau_1 - \delta \geq \tau_0$. That is, the ledger become secure again when there is honest majority (i.e., $\mathbb{A}$ holds) even if there was an interval of time when there was no honest majority (i.e., $\mathbb{A}$ did not hold). Therefore, if we consider the extreme case where $\tau_0 = 0$, we can assume without loss of generality that the ledger admits a genesis functionality parametrized by $\tau^{\mathsf{max}} = \delta$, and by m that depends on the upper bound on the number of forks that the adversary can create. Hence, there are already ledgers that might fit our requirements for $\mathcal{L}_2$, and all the advancement in the research that concerns the security of ledgers in the case of temporary dishonest majority can be used to construct good candidates of updated ledgers ($\mathcal{L}_2$) for

existing ledgers ($\mathcal{L}_1$) that can be used in our compiler.

Security for Offline Parties. Our security notion above is ensured for parties that are online during the upgrade process. Clearly it is necessary that the majority of the population's consensus-maintaining parties are honest and online, as the honest majority assumption mandates. Nevertheless, practical blockchain systems often have a large number of *consumer* parties by count who have a very small contribution to the total computational power of network, if at all, and are not significantly contributing to the maintenance of the consensus. These nodes can be wallets and other clients who mainly consume, rather than maintain, the blockchain, and are often offline for longer periods of time. Regardless, these nodes constitute the economic majority of the nodes and we must ensure they can also upgrade safely. The critical situation arises when such a party goes offline prior to an upgrade, remains offline during every phase of the upgrade, and comes online long after the rest of the population has successfully upgraded. Before describing how to construct a protocol that can protect these parties, let us briefly observe why an attack is easily possible by a minority adversary in a construction with no relevant protective mechanism. Consider a situation where a hard-fork-style change takes place and that blocks mined by upgraded parties after the upgrade are incompatible with blocks mined prior to the upgrade, i.e., after the upgrade, an unupgraded party will not consider an upgraded block as valid and an upgraded party will not consider an unupgraded block as valid. After the upgrade has been completed, the majority of the population will shift their mining power to mining new-style blocks. The adversary can take advantage of this situation to *ex post facto* attack the old system, which now remains unprotected as no significant mining power remains to secure it. As such, she can break the *common prefix* property, rewrite history, and subvert the upgrade signaling mechanism itself. More concretely, an adversary in this situation forks the old chain from the parent of the block in which upgrade information appeared for the first time and continues mining a chain parallel to the one that yielded the upgrade. As soon as that alternative history overtakes the old chain in terms of work, the adversary is successful. Any offline party who wakes up afterwards will use the old-style consensus rules to choose the blockchain and hence the upgrade will not appear in its view. The adversary has succeeded in isolating the offline party from the rest of the network. To rectify the above issue, a practical implementation of the protocol must leverage the mining power of the upgraded population to maintain *both* the new chain while at the same time securing the old chain. We propose a solution for the case where $\mathcal{L}_1$ and $\mathcal{L}_2$ are two proof-of-work or two proof-of-stake type of ledgers. Our solution leverages on a variation of 2-for-1 mining [GKL15b]. An upgraded miner works as follows. They maintain the longest chain C in view of the new protocol rules, but also the longest chain C' in the view of an unupgraded party. In case of hard fork, these two chains will differ. When they are about to mine a new block on top of the upgraded chain, they construct a new-style candidate block b extending C as usual. In addition, they also construct an empty (transactionless) old-style block b' on top of the best unupgraded chain C'. In a commentary section of the old-style candidate block b', such as the coinbase transaction, the miner places the hash $H(b)$ of the new-style candidate block. The miner then attempts to find proof-of-work for the old-style block, i.e., some nonce ctr that satisfies the proof-of-work equation $H(b' \,\|\, \mathsf{ctr}) \leq T$ for the mining target T. If such proof-of-work is found, then the block b' is broadcast to the network and adopted as the tip of the longest unupgraded chain by the rest of the (upgraded or unupgraded) miners. Note that this block is designed to be backwards-compatible in the sense that it will be accepted by unupgraded miners even though they remain unaware of the upgrade. On the other hand, if the *reverse proof-of-work equation* $H(b' \,\|\, \mathsf{ctr})^R \leq T$ is satisfied (where $H(\cdot)^R$ denotes the reversed bitstring of $H(\cdot)$), then b' and the respective proof-of-work and blocks b', b are broadcast to the network. This time unupgraded miners will not consider this a valid block. However, upgraded miners examine the validity of the block b contained within the commentary section of b' and check that the reverse proof-of-work equation is satisfied. If so, they adopt the block b as the next block in their upgraded blockchain. The above mechanism is the only mechanism by which new-style blocks

are accepted by upgraded honest miners. The protocol just described has two advantages. Firstly, the upgraded honest miners make use of their mining power to contribute to the security of both the old and the new-style chain simultaneously. Therefore, an adversary cannot attack the old chain *ex post facto*. Secondly, instead of *dividing* their mining power between the two chains, the honest parties only use their mining power *once* to mine on *both* networks, because the hash function is only evaluated once. As such, the honest mining power is not diminished by the use of this mechanism. We observe that, in the Random Oracle model, the last bits of the hash output remain uniformly distributed conditioned on the fact that the proof-of-work equation has a solution. Therefore, finding a solution of the proof-of-work equation and finding a solution of the reverse proof-of-work equation are two independent events (they will occur simultaneously so rarely that the honest parties can ignore this possibility). Lastly, note that this scheme can be used repeatedly when multiple upgrades have occurred on top of one another, simply by treating a portion of the bits of the hash as the significant bits to test against the proof-of-work equation (e.g., for a second upgrade, the hash output can be split in three equal parts to be tested against the proof-of-work equation). This scheme therefore theoretically resolves the question of securing offline parties. In practice, because the scheme adds significant implementation complexity, implementors may elect to maintain this backwards-compatibility mechanism for a limited amount of time. In that case, parties who have remained offline longer than the backwards-compatibility mechanism is maintained, will have no guarantees for security, similarly to a classical system whose long-term support window has expired. The scheme requires the added complexity of mining two blocks simultaneously only in the case of proof-of-work. This is due to the nature of proof-of-work and specifically the fact that each query counted towards the proof-of-work quota can only be devoted to a specific message. In proof-of-stake blockchains, the solution for maintaining the security of offline unupgraded parties is the obvious one and allows for a much simpler implementation: We require upgraded parties to mint, alongside their new-style blocks extending the longest upgraded chain and containing transactions, also empty old-style blocks extending the longest unupgraded chain, to ensure the security of their unupgraded counterparts.

Chapter 3

Mining for Privacy

Non-interactive zero-knowledge proofs, and more specifically succinct non-interactive zero-knowledge arguments (zk-SNARKS), have been proven to be the "Swiss army knife" of the blockchain and distributed ledger space, with a variety of applications in privacy, interoperability and scalability. Many commonly used SNARK systems rely on a *structured reference string* (SRS), the secure generation of which turns out to be their Achilles heel: If the randomness used for the generation is known, the soundness of the proof system can be broken with devastating consequences for the underlying blockchain system that utilises them. In this work we describe and analyse, for the first time, a blockchain mechanism that produces a secure SRS with the characteristic that security is shown under comparable conditions to the blockchain protocol itself. Our mechanism makes use of the recent discovery of *updateable* structured reference strings to perform this secure generation in a fully distributed manner. In this way, the SRS emanates from the normal operation of the blockchain protocol itself without the need of additional security assumptions or off-chain computation and/or verification. We provide concrete guidelines for the parameterisation of this setup which allows for the completion of a secure setup in a reasonable period of time. We also provide an incentive scheme that, when paired with the update mechanism, properly incentivises participants into contributing to secure reference string generation.

3.1 Introduction

In the domain of distributed ledgers, non-interactive zero-knowledge (NIZK) proofs have many interesting applications. In particular, they have been successfully used to introduce privacy into these inherently public peer-to-peer systems. Most notably, Zerocash [BCG+14] demonstrates their usefulness in the creation of private currencies. Beyond this, there are numerous suggestions [KMS+16, JKS16,SBG+19] to apply the same technology to smart contracts for increased privacy. Beyond privacy, other applications of zero knowledge include blockchain interoperability, e.g., [GKZ19], and scalability, e.g., [But].

For the practical efficiency of these designs, two things are paramount: The succinctness of proofs, and the speed of verifying these proofs. The distributed nature of the ledgers mandates that a large number of users store and verify each proof made, rendering many zero-knowledge proof systems not fit for purpose.

Research into so-called zk-SNARKs [PHGR13,Gro16,GM17,GKM+18,MBKM19] aims at optimising exactly these features, with proof sizes typically under a kilobyte, and verification times in the milliseconds. It is a well-known fact that non-interactive zero-knowledge requires some shared randomness, or a *common reference string*. For many succinct systems [PHGR13, Gro16, GM17, GKM+18, MBKM19], a stronger property is necessary: Not only is a shared random value needed, but it must adhere to a specific *structure*. Such structured reference strings (or SRS) typically consist of related group elements: g^{x^i} for all $i \in \mathbb{Z}_n$, for instance.

The obvious way of sampling such a reference string from public randomness reveals the exponents used – and knowledge of these values breaks the soundness of the proof system itself. To make matters worse, the security of these systems typically relies (among others) on *knowledge of exponent* assumptions, which state that to create group elements related in such a way *requires* knowing the underlying exponents and hence any SRS sampler will have to "know" the exponents used and be trusted to erase them, becoming effectively a single point of failure for the underlying system. While secure multi-party computation can be, and has been, used to reduce the trust placed on such a setup process [Zca18], the selection of the participants for the secure computation and the verification of the generation of the SRS by the MPC protocol retain an element of centralisation. Using an MPC setup remains a controversial element in the setup of a decentralised system that requires SNARKs.

Recent work has found succinct zero-knowledge proof systems with *updateable* reference strings [GKM+18, MBKM19]. In these systems, given a reference string, it is possible to produce an updated reference string, such that knowing the trapdoor of the new string requires both knowing the trapdoor of the old string, *and* knowing the randomness used in the update. [GKM+18] conjectured that a blockchain protocol may be used to securely generate such a reference string. Nevertheless, the exact blockchain mechanism that produces the SRS and the description of the security guarantees it can offer has, so far, remained elusive.

3.1.1 Our Contributions

In this work we describe and analyse, for the first time, a blockchain mechanism that produces a secure SRS with the characteristic that security is shown for similar conditions under which the blockchain protocol is proven to be secure. Notably different, we make implicit use of secure erasure, and require honest majority only during a specific initialisation period. The SRS then emanates from the normal operation of the blockchain protocol itself without the need of additional security assumptions or off-chain computation and/or verification.

We rely primarily on the *chain quality* property of "Nakamoto-style" ledgers [GKL15b] – distributed ledgers in which a randomised process selects which user may append a block to an already established chain. Such ledgers rely on an honest majority of hashing power (or some other resource) – and can be shown to guarantee a chain quality property which suggests that any sufficiently long chain segment will have some blocks created by an honest user, cf. [GKL15b, PSs17, GKL17].

Our construction, described in section 3.3 integrates reference string updates into the block creation process, but we face additional difficulties due to update calculation being a computationally heavy operation (albeit, contrary to brute-force hashing, useful). The issues arising from this are twofold. Firstly, an adversarial party can take shortcuts by supplying a low amount of entropy in their updates, and try to utilise this additional mining power to subvert the reference string which potentially has a large benefit for the adversary. Secondly, even non-colluding rational block creators may be incentivised to use bad randomness which would reduce or remove any security benefits of the updates. Our work addresses both of these issues.

We prove formally that our mechanism produces a secure reference string in section 3.12 by providing an analysis in the universal composition framework [Can01]. Furthermore, in section 3.4, we demonstrate via experimental analysis how to concretely parameterise a proof-of-work ledger to ensure that an adversary which takes shortcuts (while honest users do not) will still fail in subverting the reference string. The concrete results provided in our experimental section can be used to inform the selection of parameters in order to run our reference string generation mechanism in live blockchain systems.

We further introduce an incentive scheme in section 3.5, which ensures that rational participants in the protocol, who intend to maximise their profits, will avoid low-entropy attacks. In short, the incentive mechanism mandates that a random fraction of update contributors in the final chain will be asked to reveal their trapdoor, which will be verified to be the output of a random oracle by the

underlying ledger rules. Only if a user can demonstrate that their update is indeed random do they receive a suitably determined reward for their effort. Careful choice of the reward assignment enables us to demonstrate that rational participants will utilise high entropy exponents, thus contributing to the SRS computation.

3.1.2 Related Work

Beyond the obvious relation to the works introducing updateable reference strings in [GKM+18,MBKM19] (most notably Sonic [MBKM19], which we follow closely in our instantiation in section 3.7), there have been attempts of practically answering the question of how to securely generate reference strings. These have been in a setting where the string is *not* updateable.

Notably [BGG19] describes the mechanism used by Sprout, the first version of Zcash, during the initial setup of the cryptocurrency's SRS. It uses multi-party computation to generate a reference string, with a root of trust on the initial group of people participating. Due to performance constraints on the MPC protocol, the set of parties participating is relatively small, although only the honesty of a single participating party is required.

For the Sapling version of Zcash, a different approach was used when their reference string was replaced (due to an upgrade of the zero-knowledge statement, and proof system used). Their second CRS generation mechanism, described in [BGM17] uses a multiple-phase round-robin mechanism to generate a reference string for Groth's zk-SNARK [Gro16]. They utilise a random beacon to ensure the uniform distribution of the result, and a coordinator to perform deterministic auxiliary computations.

A great deal of work has also gone into the design of non-interactive zero-knowledge which does not require structure in its references, such as DARK [BFS20], STARKs [BBHR18], and Bulletproofs [BBB+18a]. While these pose a promising alternative which does not require the techniques used in this work, leveraging updatability of reference strings may permit greater efficiency without additional security assumptions, and may be useful in instantiating generic constructions, such as the polynomial commitments-based Halo Infinite [BDFG20].

3.2 Updateable Structured Reference Strings

While updateable structured reference strings (uSRSs) are modelled in the works we are building on [MBKM19, Section 3.2], we model their security in the setting of universal composability (UC) [Can01]. Here, a uSRS is a reference string with an underlying trapdoor τ, which has had a structure function S imposed on it. $S(\tau)$ is the reference string itself, while τ is not revealed to the adversary. In section 3.7, we prove that Sonic [MBKM19] (with small modifications for extraction, as described in subsection 3.2.2), satisfies all the properties we require in this section. Our main proof is independent of the Sonic protocol however, and applies to any updateable reference string scheme satisfying the properties laid out in the rest of this section.

3.2.1 Standard Requirements

A uSRS scheme $\mathcal{S}$ consists of a trapdoor domain T, an initial trapdoor τ_0, a set P of permissible (and invertible) permutations over T (i.e. bijective functions whose domain and codomain is T), and a structure function S with the domain T. We require P to include the identity function id, and to be closed under function composition: $\forall p_1, p_2 \in P : p_1 \circ p_2 \in P$. An efficient permutation lifting $\dagger$ should exist, such that for any permutation $p \in P$ and $\tau \in T$, $p^\dagger(S(\tau)) = S(p(\tau))$. Finally, there must exist algorithms $\rho \leftarrow \mathsf{ProveUpd}(S(\tau), p)$ and $b \leftarrow \mathsf{VerifyUpd}(S(\tau), \rho, S(p(\tau)))$ for creating and verifying update proofs respectively. The format of these update proofs is not specified, however the following constraints must be met:

1. **Correctness.** Applying an honestly generated update proof will verify: $\forall p \in P, \tau \in T$: $\mathsf{VerifyUpd}(S(\tau), \mathsf{ProveUpd}(S(\tau), p), S(p(\tau)))$.
2. **Structure preservation.** Applying *any* valid update is equivalent to applying *some* permutation $p \in P$ on the trapdoor: $\forall \rho, \tau, \mathsf{srs}' : \mathsf{VerifyUpd}(S(\tau), \rho, \mathsf{srs}') \implies \exists p \in P : \mathsf{srs}' = S(p(\tau))$.
3. **Update uniformity.** Applying a random permutation is equivalent to selecting a new random trapdoor: Let D be the uniform distribution over T, and for all $\tau \in T$, let D_τ be the uniform distribution over the multiset $\{\ p(\tau) \mid p \in P\ \}$. Then $\forall \tau \in T : D = D_\tau$.

We define a corresponding UC functionality $\mathcal{F}_{\mathsf{uSRS}}$, which provides a reference string $S(p(\tau_{\mathcal{H}}))$, which the adversary can influence by providing the permutation $p \in P$, given only $S(\tau_{\mathcal{H}})$ as input, for a randomly sampled $\tau_{\mathcal{H}} \in T$.

Functionality $\mathcal{F}_{\mathsf{uSRS}}$

The updateable structured reference string functionality $\mathcal{F}_{\mathsf{uSRS}}$ allows the adversary to update a reference string by applying a permutation from a set of permissible permutations P.

The functionality is parameterised by a trapdoor domain T, a structure function S, and a set of permissible permutations P over T.

State variables and initialisation values:

Variable	Description
$\tau_{\mathcal{H}} := \bot$	The honest part of the trapdoor
$\tau := \bot$	The trapdoor

When receiving a message HONEST-SRS *from* $\mathcal{A}$*:*

if $\tau_{\mathcal{H}} = \bot$ **then let** $\tau_{\mathcal{H}} \xleftarrow{R} T$
return $S(\tau_{\mathcal{H}})$

When receiving a message SRS *from a party* ϕ*:*

query $\mathcal{A}$ **with** (PERMUTE, ϕ) and **receive the reply** p
if $\tau = \bot$ **then**
  assert $p \in P \wedge \tau_{\mathcal{H}} \neq \bot$
  let $\tau \leftarrow p(\tau_{\mathcal{H}})$
return $S(\tau)$

We believe this functionality to be of independent interest, and it is not explicitly tied to our implementation. Notably, while we use a distributed ledger as a weak form of a broadcast channel, other broadcasts can be considered without modification to this functionality. While, as presented, the functionality does not dictate any specific usage, we conjecture that when parameterised with an appropriate structure function and permutation set it can be used to securely instantiate updateable SRS-based SNARKs, such as Sonic [MBKM19], Marlin [CHM+20], or Plonk [GWC19]. Due to the UC setting, this would require additional lifting to enable UC knowledge extraction, such as that of C∅C∅ [KZM+15].

3.2.2 Simulation Requirements

In addition to the basic properties of correctness, structure preservation, and update uniformity, any simulator wishing to help realise $\mathcal{F}_{\mathsf{uSRS}}$ via updates will need to have access to two additional properties:

1. **Update proof simulation.** From an initial SRS $S(\tau)$ for which the simulator knows the trapdoor, it can produce a valid update to any (correctly structured) SRS. Formally: $\exists \mathcal{S}_\rho \forall \tau_1, \tau_2 \in T$: $\mathsf{VerifyUpd}(S(\tau_1), \mathcal{S}_\rho(\tau_1, S(\tau_2)), S(\tau_2))$, where $\mathcal{S}_\rho$ is a PPT algorithm.

2. **Permutation extraction.** The simulator must be capable of extracting the permutation p underlying any valid adversarial update proof.

The most natural method to achieve permutation extraction would be using white-box extractors, as the updates themselves typically rely on some form of knowledge assumption, such as knowledge-of-exponent. However, white-box extractors cannot be used in UC proofs. Instead, we will assume that the update proof is proven to correspond to a specific trapdoor through a lower-level NIZK. Crucially, this lower-level NIZK should not require a *structured* reference string, and rely only on a common random string, or a random oracle. Fortunately, it is not subject to stringent efficiency requirements as section 3.4 demonstrates.

Specifically, we assume that the basic update proof ρ is a statement in a NIZK relation $\mathcal{R}$ where the witness is an encoding of the corresponding permutation p. We require each update proof to have one and only one corresponding permutation, formally expressed by requiring $\mathcal{R}$ to be a bijection. This results in a straightforward modification to the $\mathsf{ProveUpd}$ and $\mathsf{VerifyUpd}$ algorithms that permits the extraction of the underlying permutations even in the UC setting: $\mathsf{ProveUpd}$ also creates a NIZK proof π of (ρ, p), and returns (ρ, π), While $\mathsf{VerifyUpd}$ returns true only if this newly embedded NIZK proof also verifies.

The addition of this NIZK trivially preserves all security properties including correctness, due to the definition of $\mathcal{R}$:

Definition 4. *A uSRS scheme is permutation extractable if the relation*

$$\mathcal{R} := \{(\mathsf{ProveUpd}(S(\tau), p), p) \mid \tau \in T, p \in P\}$$

is a bijection, and in NP.

We show in section 3.7 that the relation required for the case of Sonic [MBKM19] can be efficiently constructed, and leave the question of how to achieve extraction without the reliance on a further NIZK to future work.

3.3 Building uSRS from Chain Quality

This section shows how to securely initialise a uSRS using a distributed ledger by requiring block creators to perform updates on an evolving uSRS during an initial setup period. After waiting for agreement on the final uSRS, it can be safely used. To formally model this approach, we discuss the ideal and real worlds used in our simulation proof. Both worlds have access to a ledger, however, the ideal world's ledger is independent of the reference string (which is instead provided by the independent $\mathcal{F}_{\mathsf{uSRS}}$ functionality), while the real world's ledger is programmed to generate it using updates.

3.3.1 High-Level Overview

This basic premise of this chapter relies on Nakamoto-style ledgers' basic means of operation: Different users can extend a chain of blocks if they can satisfy some condition, with this condition being associated with a type of hardness which ensures attackers are limited in the number of extensions they can perform. Given such a structure, we associate a uSRS update with each block prior to a time δ_1. This time is selected such that the security properties of the ledger ensure at least one of the blocks is honest in each competitive chain at this point.

In our modelling, we construct this from a ledger functionality with an additional *leadership state*, which is derived from information miners embed in their blocks. Specifically for our case, these encode uSRS updates. We leave this sufficiently general to allow other uses as well. The basic idea is to show that a ledger which performs uSRS updates in its leadership state is equivalent to one which doesn't,

but is accompanied by the $\mathcal{F}_{\mathsf{uSRS}}$ functionality. They make up our real and ideal worlds respectively. After time δ_1, users wait a further time period δ_2 until common prefix ensures that all parties agree on the reference string.

While ledger functionalities are often treated as global, our approach effectively constructs one ledger from another – the ledger is not a dependency of our protocol, but a component. In this context, globality is irrelevant, as the environment already has direct access to the functionality. We expect protocols building on the ledger to use it in a global fashion, however. The same is not true for the uSRS – most usages will likely rely on the simulator being able to extract its trapdoor.

3.3.2 Our Ledger Abstraction

Our construction of the updateable structured reference string functionality relies heavily on the properties of *common prefix*, *chain quality*, and *chain growth* defined in the "Bitcoin backbone" analysis by Garay et al. [GKL15b], for Nakamoto-style consensus algorithms. Despite our use in the section title, we make use of all three properties, not just that of chain quality. We emphasise chain quality, as it is the property central to ensuring an honest update has occurred. We briefly and informally restate the three properties:

- **Common prefix.** Given the current chains Π_1 and Π_2 of two parties, and removing k blocks from the first, it is a prefix of the second: $\Pi_1^{\lceil k} \prec \Pi_2$.
- **Chain quality.** For any party's current chain Π, any consecutive l blocks in this chain will include μ blocks created by an honest party.
- **Chain growth.** If a party's chain is of length c, then s time slots later, it will be at least of length $c + \gamma$.

These parameters determine the length of the two phases of our protocol. In the first phase, we construct the reference string itself from the liveness parameter (assuming $\mu \geq 1$), and in the second phase, we wait until this reference string has propagated to all users. The length of the first phase is at least $\delta_1 \geq \lceil l\gamma^{-1} \rceil s$, and that of the second at least $\delta_2 \geq \lceil k\gamma^{-1} \rceil s$. Combined, they make up the total uSRS generation delay $\delta \geq (\lceil l\gamma^{-1} \rceil + \lceil k\gamma^{-1} \rceil)s$.

We assume a ledger which guarantees the backbone properties, formally described in subsection 3.8.1. While we do not prove any specific existing proof-of-work ledger (or those based on a different leader-selection mechanism) formally UC-realise this specific formalisation, we argue all ledgers with "Nakamoto-style" (as opposed to BFT-style) consensus do so. in subsection 3.8.2 Our functionality further depends on a *global clock* $\mathcal{G}_{\mathsf{clock}}$, defined in subsection 3.11.1. For the purposes of this chapter, it is sufficient that this is a beacon providing monotonically increasing values representing the current time to any party requesting them.

In addition to this, we assume each block created can contain additional information, provided by its creator (the "miner"), which can be aggregated to construct a "leader state". Each created block is associated with an *update* a, and the ledger is parameterised by two procedures, Gen, and Apply, which describe the honest selection of updates, and the semantics of updates respectively. Looking forward, these utilise $\mathsf{ProveUpd}$ and $\mathsf{VerifyUpd}$ internally, although the formalism is sufficiently general to allow usage of the leader state for other, parallel purposes. The exact parameters differ in our ideal and real world, with the ideal world "hiding" the uSRS updates. Additionally, the real world adds time-sensitivity: It does nothing to the SRS after the setup period. Gen is randomised, takes a leader state σ and the current time t as inputs, and produces an update a. Apply takes a leader state σ, an update a, and an update time t, and returns a successor state σ': $\sigma' = \mathsf{Apply}(\sigma, (a, t))$. For a chain, the leader state may be computed by sequentially applying all updates in the chain, starting from an initial state $\varnothing$.

The adversary controls when and which party creates a new block, as well as the transactions each new block contains (provided it does not violate the backbone properties). For transactions created by a corrupted party, the adversary can further control the block's timestamp (within the reasonable limits of not being in the future, and being after the previous block), and the desired update a itself. For honest parties updates, Gen is used instead.
The UC interfaces our ledger provides are:

- SUBMIT. Submitting new transactions for the ledger.
- READ. Reading the confirmed sequence of transactions.
- PROJECTION. Reading the current chain's sequence of (potentially unconfirmed) transactions.
- LEADER-STATE. Reading the confirmed leader state.
- ADVANCE. The adversary switches a party to a longer chain.
- EXTEND. The adversary instructs a party to create a block.

While this ledger abstraction is not the focus of this chapter, we believe it to be of independent interest in cases where finer control over miner's actions, or better access to the competing chains is desired.

3.3.3 The Ideal World

Our ideal world consists of two functionalities, composed in parallel (by which we mean: the environment may address either, and they do not interact). The first is a variant of $\mathcal{F}_{\mathsf{uSRS}}$, with the modification that it cannot be addressed by honest parties before δ time slots have passed. Formally, this modification is made with a wrapper functionality $\mathcal{W}_{\mathsf{delay}}(\mathcal{F}, \delta)$, described in subsection 3.11.4.

The second is the Nakamoto-style ledger functionality, parameterised with arbitrary leader-state generation and application procedures which are also partially used in the hybrid world: $\mathsf{Gen} = \mathsf{GenIdeal}$ and $\mathsf{Apply} = \mathsf{ApplyIdeal}$, and the following ledger parameters:

1. A common prefix parameter k.
2. Chain quality parameters μ and l.
3. Chain growth parameters γ and s.

Formally then, our ideal world consists of the pair $(\mathcal{W}_{\mathsf{delay}}(\delta, \mathcal{F}_{\mathsf{uSRS}}), \mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}})$, as well as the global functionality $\mathcal{G}_{\mathsf{clock}}$.

3.3.4 The Hybrid World

In our hybrid world, we use a uSRS scheme $\mathcal{S}$, with algorithms $\mathsf{ProveUpd}$, $\mathsf{VerifyUpd}$, the structure function S, permissible permutations P, permutation lifting $\dagger$, initial trapdoor τ_0. The hybrid world consists of a separate Nakamoto-style ledger $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$, a NIZK functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, and the global clock $\mathcal{G}_{\mathsf{clock}}$. The ledger is then parameterised by the same chain parameters as those in the ideal world, and the following leader-state procedures:

procedure $\mathsf{Apply}((\mathsf{srs}, \sigma^{\mathsf{ideal}}), ((\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}), t))$
  if $\mathsf{srs} = \varnothing$ **then let** $\mathsf{srs} \leftarrow S(\tau_0)$
  if $t \leq \delta_1 \wedge \mathsf{VerifyUpd}(\mathsf{srs}, \rho, \mathsf{srs}')$ **then**
    send $(\text{VERIFY}, \rho, \pi)$ **to** $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ and **receive the reply** b
    if b **then**
      let $\mathsf{srs} \leftarrow \mathsf{srs}'$
  return $(\mathsf{srs}, \mathsf{ApplyIdeal}(\sigma^{\mathsf{ideal}}, a^{\mathsf{ideal}}, t))$

procedure $\mathsf{Gen}((\mathsf{srs}, \sigma^{\mathsf{ideal}}), t)$
  if $t > \delta_1$ **then**
    return $(\epsilon, \epsilon, \epsilon, \mathsf{GenIdeal}(\sigma^{\mathsf{ideal}}, t))$
  else
    let $p \xleftarrow{R} P; \rho \leftarrow \mathsf{ProveUpd}(\mathsf{srs}, p)$
    send (PROVE, ρ, p) **to** $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ and **receive the reply** π
    return $(p^{\dagger}(\mathsf{srs}), \rho, \pi, \mathsf{GenIdeal}(\sigma^{\mathsf{ideal}}, t))$

Note that these parameterising algorithms use $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, and are therefore the reason the ledger depends on this hybrid functionality.

Key here is that once a block is received after the initial chain quality period, any reference string update it may declare is no longer carried out – at this point the uSRS is not necessarily stable, as the chain may still be reorganised, but should not change for this particular chain. Further, these procedures always mimic the ideal-world behaviour, extending it rather than replacing it. This demonstrates the composability of allowing block leaders to produce updates: One system using updates for security does not impact other parallel uses of the leadership state.

There is little additional work to be done to UC-emulate the ideal-world behaviour, besides ensuring that queries are routed appropriately, especially how the reference string is queried in the hybrid world. We describe this with a small "adaptor" protocol in section 3.9, LEDGER-ADAPTOR. This forwards most queries, and treats uSRS queries as querying the appropriate part of the leader state after time δ, and by ignoring them before. Formally, our real world consists of the global clock $\mathcal{G}_{\mathsf{clock}}$, and the system $\text{LEDGER-ADAPTOR}(\delta, \mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}(\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}))$.

3.3.5 Alternative Usage of $\mathcal{G}_{\mathsf{clock}}$

In both worlds, $\mathcal{G}_{\mathsf{clock}}$ is used to determine the cutoff point after which the reference string is deemed secure. A simple alternative to this usage of the clock is to instead rely on the length of the chain for this purpose. We did not make this choice as it complicates the ideal world: The delay wrapper would have to communicate with the ideal world ledger, and query it for the length of parties' chains. We do not regard a clock as a significant additional assumption, however, little of the remainder of this chapter differs if chain lengths are used instead. Even in this case, a clock is present to guarantee liveness, although it is used only to constrain the adversary.

3.3.6 UC Emulation

Our security is derived through UC-emulation, stated in the following theorem:

Theorem 3. *For any updateable reference string scheme $\mathcal{S}$, satisfying correctness, structure preservation, update uniformity, update simulation with $\mathcal{S}_\rho$, and permutation extraction,* LEDGER-ADAPTOR *(in the $(\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}, \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}})$-hybrid world, parameterised as in subsection 3.3.4) UC-emulates the pair of functionalities $(\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}, \mathcal{W}_{\mathsf{delay}}(\delta, \mathcal{F}_{\mathsf{uSRS}}))$, parameterised as in subsection 3.3.3, in the presence of the global clock functionality $\mathcal{G}_{\mathsf{clock}}$, with the simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$.*

A full security proof of Theorem 3 may be found in section 3.12, and the simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$ may be found in section 3.10.

3.4 Implementation and Parameter Selection

We have implemented [Ker20] Sonic's update mechanism (described in section 3.7), and using this provide performance estimates for SRS generation in a live blockchain network. Further, we simulate the optimal adversarial attack strategy, and demonstrate how this may be used to select optimal parameters

for the secure generation of reference strings. We demonstrate that for currently typical applications, these parameters are practical for real-world usage.

While we have not modified a full blockchain client to utilise this extended consensus, we discuss the impact it would have on each of the following points:

- block verification
- block generation
- chain reorganisation
- network usage
- local storage

While the Bitcoin backbone paper [GKL15b] provides bounds on chain parameters in given situations, these have three main drawbacks in the context of this chapter:

1. The bounds are not tight.
2. The criteria for security is stricter than required: It asserts liveness and persistence are never violated, while this chapter only requires them in a few select cases.
3. The analysis is in the synchronous model – while the generation and verification of reference strings can take a significant amount of time.

To obtain sensible parameters to generate reference strings, we measure the time taken for computing and verifying updates, and factor this processing overhead into a simulation of the optimal adversarial strategy to subvert the SRS generation procedure.

The implementation and numbers provided for execution time and storage use the commonly used BLS12-381 curve pair. Circuits which have been practically deployed tend to require a depth of at most half a million, so we will often assume a Sonic uSRS depth of 500,000. All data shown is available at [Ker20], and may be reproduced with the provided source code.

3.4.1 Execution Time of uSRS Operations

We tested our implementation of the uSRS generation mechanism on an AMD Ryzen 7 2700X 8-core processor with hyper-threading enabled. This processor is a standard consumer-grade CPU – in proof-of-work mining it is likely that miners will have access to better hardware. All operations have been parallelised, and the verification operation has been additionally optimised to use less pairing operations. The workload, especially for uSRS generation, is also highly parallelisable (consisting of primarily a large number of group exponentiations), suggesting further improvements by utilising GPUs and clusters of machines are possible. If such improvements are applied, the total time *delay* required for the secure generation procedure, as well as the optimal intended block time could be reduced proportionally to the increase in parallelisation; assuming paralellisation across 10 machines could reduce both by an order of magnitude, for instance.

We measured the time taken for create and verify a uSRS update in relation to the uSRS depth in Figure 3.1. For our NIZK, we use a UC-secure Fischlin proof, described in subsection 3.7.3. We measure the overhead of these proofs to be 23.956ms for proving and 1.567ms for verifying (a Fiat-Shamir proof of the same type was measured to 0.921ms and 0.870ms respectively), using SHA-3 in place of a random oracle. For larger dimensions of reference strings, neither have much impact on the total runtime.

Finally, we implemented *aggregate updates*: The bulk of Sonic's update verification procedure is concerned with verifying the structure of the reference string, while a few parts of it verify that it is an exponentiation of the previous string. By retaining only the latter parts, a series of updates can be

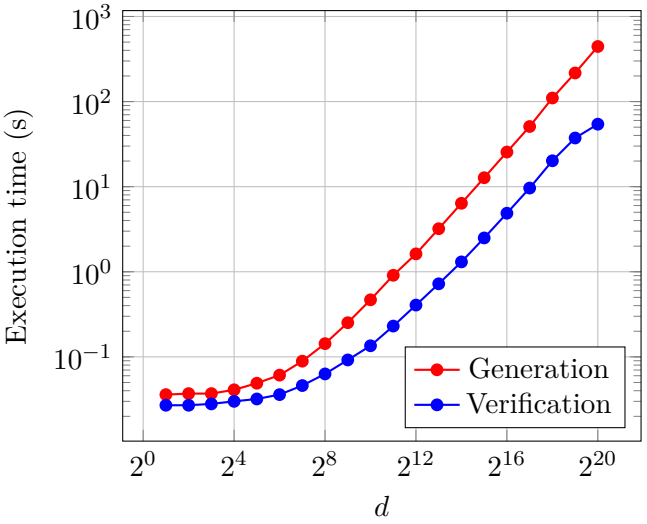


Figure 3.1: The time taken to produce and verify uSRS updates.

verified almost as quickly as a single update. The verification of aggregate proofs has an overhead of 1.634ms per update included in the aggregate. The bulk of this cost arises from the verification of the Fischlin proof. This allows for even large chain reorganisations to be quickly verified.

3.4.2 Simulating the Optimal Attack Strategy

The mechanism we have presented in this chapter operates in two phases. In the first phase, the adversary has the chance to *subvert* the reference string, while in the second phase it can carry out a denial of service attack, potentially convincing users that an incorrect (but not subverted) reference string is the canonical one.

For the first phase, the adversary's optimal strategy is to mine entirely independently from any honest activity: the adversary cannot adopt any honest block – doing so would break the subversion of its reference string. Further, the adversary has no reason to share any of its own blocks except if it reached the threshold of having a fully valid subverted reference string – it only gives the honest network a chance to catch up, in the case that the adversary is ahead. This allows for a straightforward simulation of the consensus protocol: The probability of either honest parties, or the adversary creating an individual block is exponentially distributed. In addition to this, honest parties have a fixed processing overhead before they may start mining: This may include a networking delay, but more crucially it includes the time taken to verify a newly received block's uSRS update, and to produce the subsequent update. We assume that the adversary can bypass large parts of this overhead, by virtue of network dominance, by skipping verification, and by producing reference string updates with small (and therefore insecure) exponents.

The overhead manifests as shifting the honest party's exponential distribution for block generation by a fixed constant. More precisely, we parameterise each experiment by:

- The intended time between blocks b
- The combined networking, and update overhead d
- The fraction of adversarial mining power α

Of these three, d can be seen as fixed, depending on the depth of the uSRS being generated, and the corresponding speed of verification and update generation. For simplicity, we assume a uSRS depth of

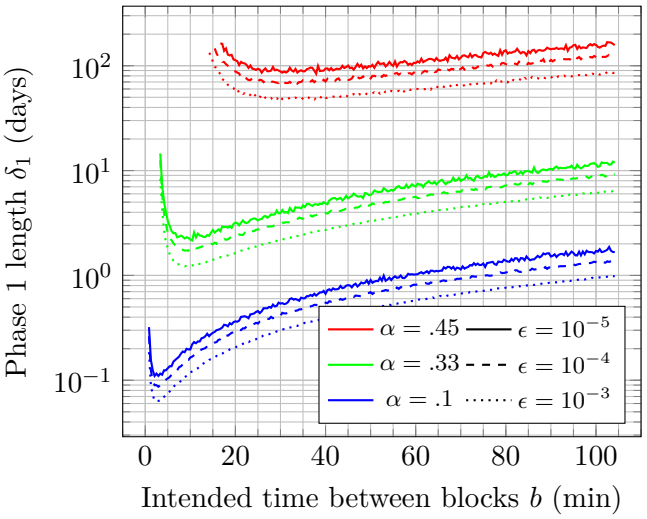


Figure 3.2: The time required to generate a secure uSRS, as a function of the intended time between blocks. This depends on the proportion of adversarial mining power α, and the bound ϵ on the probability of subversion. Each data point represents the time until at most a fraction of ϵ of one million parallel experiments ended in adversarial victory. Values are given assuming $d = 250s$, and both axes scale linearly to d.

500,000, which corresponds to d being approximately 250 seconds on our single-CPU setup.

We draw the time of the next adversarial block from the exponential distribution with $\lambda = \alpha/b$, and the next honest block from the exponential distribution with $\lambda = (1-\alpha)/b$, shifted to the right by d (i.e. the probability density is 0 for $x < d$). The simulation is then advanced to the lesser of the two times, which is resampled from the same distribution. The number of times the adversary or the honest parties have extended their chain is counted, and the honest parties win at any point if and only if the honest chain is longer than the adversarial chain.

We run one million experiments in parallel, either up to a fixed end time, or until a large enough fraction of the experiments end in honest victory. We refer to the probability of an adversarial success as the probability of subversion ϵ. Figure 3.2 demonstrates that for a fixed d, a tradeoff exists between the target time between blocks b, and the time until any given subversion threshold ϵ is met.

A practical limit of this simulation approach is that it cannot by itself determine the length of time needed to wait until ϵ is negligible for most typical security parameters. We can however observe that for fixed parameters, ϵ decreases approximately exponentially as time passes, as seen in Figure 3.3, outside of a brief initial window.

While the second phase – that where the adversary attempts to create disagreement as to which reference string is the canonical one – may initially seem different, its optimal strategy is identical, as it essentially wishes to create as long as possible a fork, starting one block prior to the end of the first phase (to select a different reference string). As creating the longest fork *forking at this point* does not allow the adversary to accept honest blocks after it, nor gives the adversary a reason to share its blocks, the adversarial strategy – and this analysis – is the same.

3.4.3 Storage and Network Usage

A Sonic reference string consists of $4d+1$ elements in $\mathbb{G}_1$ and $4d+2$ elements in $\mathbb{G}_2$. For the commonly used BLS12-381 curve pair, $\mathbb{G}_1$ elements have a storage requirement of 48 bytes each, and $\mathbb{G}_2$ elements of 96 bytes each. An update proof includes an additional two $\mathbb{G}_1$ elements, and a Fischlin proof, which

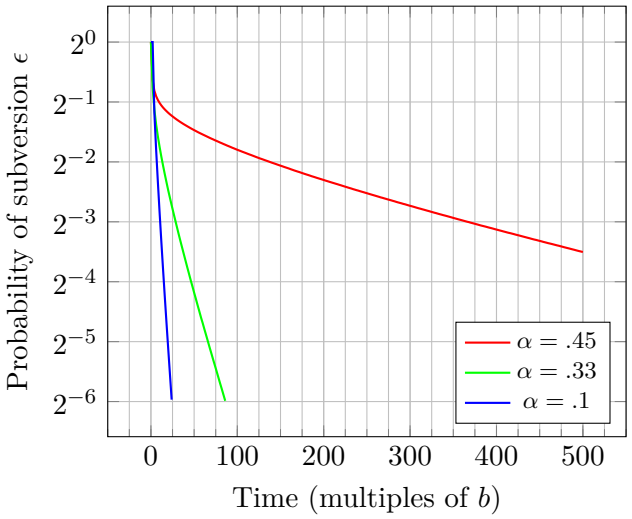


Figure 3.3: The probability of the reference string being subverted ϵ, as a function of the time passed, in multiples of the intended time between blocks b. This depends on the proportion of adversarial mining power α, and the compound overhead d. b is selected to be approximately at the minimum seen in Figure 3.2, with $d = .15b$, $d = .4b$, and $d = 2b$ for the $\alpha = .45$, .33, and .1 respectively.

itself consists of twelve iterations, each with 2 elements in $\mathbb{F}_q^*$ (each of which requires 32 bytes to store), two elements of $\mathbb{G}_1$, and a 16-bit nonce. Each part of an aggregate update has an additional two $\mathbb{G}_2$ elements.

As it is not necessary to retain intermediate reference strings, and aggregate updates are sufficient, for a chain of length l, and with an uSRS depth of d, this is a storage requirement of $576d + 288$ bytes for the uSRS itself, and $l \cdot (2 \cdot 48 + 2 \cdot 96 + 12 \cdot (2 \cdot 32 + 2 \cdot 48 + 2)) = 2,232l$ bytes for storing updates.

For 500,000 gates and chains of length 20,000, this corresponds to a total storage requirement of 318MiB, with the reference string itself being the largest part, at 275MiB. Although this is quite manageable as a storage requirement, it must be considered that the SRS itself (and a single update of around 2KiB) has to be re-transmitted with each block. While at the common home-internet upload speed of 10Mb/s, a block would take slightly under 4 minutes to transmit, it is reasonable to assume that miners would invest in high-grade connections to offset the chance of their block being replaced with a competitor. Speeds up to 10Gb/s are commercially available, which would reduce the transmission time to under a second.

One remaining issue is that of denial-of-service. The receipt and verification of a reference string is costly, and should therefore only be done *after* a block's proof-of-work has been received, which should depend on a commitment to the subsequently sent reference string – such as the update proof itself. An attacker can still perform a limited denial of service attack with blocks they legitimately mined – however this uses no more resources in verification than a legitimate block would.

3.4.4 Conclusion

Figure 3.2 provides insight into the space of tradeoffs which can be made for the secure generation of reference strings. While the secure generation of a reference string is possible even for a small honest majority, the time required to do so is much higher than for a more relaxed setting, with δ_1 being approximately three *months* for $\alpha = .45$, in contrast to around two *days* for $\alpha = .33$. The full setup is double this: six months for $\alpha = .45$, and four days for $\alpha = .33$. Perhaps surprisingly, the desired probability of subversion ϵ has a more muted effect on the required setup time.

The minima observed for δ_1 suggest that simply deploying this system on existing blockchain systems

as they are currently parameterised is unwise: Most blockchains emphasise small values of b to enable transactions to settle quickly, with even notoriously slow chains such as Bitcoin having values on the lower end of our scale. This is directly linked to the compound overhead of verification and update generation – when b is small, the adversary can better use its advantage of bypassing large parts of the verification and update procedure. As previously noted, there is a lot of room for speedup by assuming miners use greater computation power – if each miner used ten machines, even the $\alpha = .45$ case would be reduced to under a month in total.

3.5 Low-Entropy Update Mitigation

While our analysis indicates that in a Byzantine, honest majority setting, our protocol produces a trustworthy reference string, it also asks participants to dedicate computational resources to updates. It follows that in a rational setting, players need to be properly incentivised to follow the protocol. We emphasise that the rational setting is not the focus of this chapter, and optimistically, in a setting where the majority of miners are rational and a small fraction honest, the few honest blocks are sufficient to eliminate the issue described in this section.

For Sonic, a protocol deviation exists that breaks the security of the reference string: By choosing the exponent in a specific low-entropy fashion, (e.g., $y = 2^l$) the computation of the update, which primarily relies on repeated squaring, can be done significantly faster. More generally, some permutations in P may be more efficiently computable. In more detail, instead of using a random permutation p, a specific choice is made that eases the computation of srs' – in the most extreme case, for any uSRS scheme, the update for $p = \mathsf{id}$ is trivial.

3.5.1 Proposed Construction

In order to facilitate a mitigation for this class of attacks, we will need to assume an additional property of the underlying ledger, in particular it must provide a "resettable" randomness beacon: With each ADVANCE operation (where adversary must be restricted in how often it may do such ADVANCE queries), a random beacon value is sampled in a variable bcn and is associated with the corresponding block. Beacons of this kind are often easily available, for instance by hashing the proof-of-work [BCG15], and are inherent in many proof-of-stake designs. Prior work [DGKR18] demonstrates that such beacon values allow for the adversary to bias them only by "resetting" it at most a certain number of times, say t, before they are fixed by entering the ledger's confirmed state, with the exact value of t depending on the chain parameters.

We can then amend Gen to derive its random values from the random oracle, by sending the query $(\mathsf{bcn}, \mathsf{nonce})$ to $\mathcal{F}_{\mathsf{RO}}$, where nonce is a randomly selected nonce, and bcn is the previous block's beacon value. The response is used to index the set of trapdoor permutations P, choosing the result p, and the nonce is stored by miners locally, and kept private. We adapt the Phase 1 period δ_1 so that at least $l' := l(1-\theta)^{-1} + c$ blocks will be produced, where θ and c are new security parameters (to be discussed below). Next, after Phase 2 ends, we can be sure that the beacon value associated with the end of Phase 1 has been reset at most t times.

We extract from bcn l' biased coins, each with probability θ. For each block, if the corresponding coin is 1, it is required to reveal its randomness within a period of time at least as long as the liveness parameter. Specifically, a party which created one of the selected blocks may reveal its nonce. If its update matches this nonce, the party receives an additional reward of value R times the standard block reward.

While this requires a stricter chain quality property, with the ledger functionality instead enforcing that one of these l non-opened updates are honest, we sketch why this property still holds in the next section.

3.5.2 Security Intuition

Consider now a rational miner with hashing power α. We know that, at best, using an underlying blockchain like Bitcoin, the relative rewards such a miner may expect are at most $\alpha/(1-\alpha)$ in expectation; this assumes a selfish mining strategy that wins all network races against the other rational participants. Now consider a miner who uses low entropy exponents to save on computational power on created blocks and, as a result, boosts their hashing power α to an increased relative hashing power of $\alpha' > \alpha$. The attacker can further try to influence the blockchain by forking and selectively disclosing blocks which has the effect of resetting the bcn value to a preferred one. To see that the impact of this is minimal, we prove the following lemma.

Lemma 1. *Consider a mapping $\rho \mapsto \{0,1\}^{l'}$ that generates l' independent biased coin flips, each with probability θ, when ρ is uniformly selected. Consider any fixed $n \leq l'$ positions and suppose an adversary gets to choose any one out of t independent draws of the mapping's random input with the intention to increase the number of successes in the n positions. The probability of obtaining more than $n(1+\epsilon)\theta$ successes is $\exp(-\Omega(\epsilon^2\theta n) + \ln t)$.*

Proof. In case $t = 1$, result follows from a Chernoff bound on the event E defined as obtaining more than $n(1+\epsilon)\theta$ successes, and has probability $\exp(-\Omega(\epsilon^2\theta n))$. Given that each reset is an independent draw of the same experiment, by applying a union bound we obtain the lemma's statement. □ □

The optimal strategy of a miner utilising low-entropy attacks is to minimise the number of blocks of other miners are chosen, to increase its relative reward. Lemma 1 demonstrates that at most a factor of $(1+\epsilon)^{-1}$ damage can be done in this way. Regardless of whether a miner utilises low-entropy attacks or not, their optimal strategy beyond this is selfish mining, in the low-entropy attack mining in expectation $l'\alpha'/(1-\alpha')$ blocks [GKL15b]. A rational miner utilising low-entropy attacks will not gain any additional rewards, while a miner not doing so will gain at least $l'\alpha/(1-\alpha)(1+\epsilon)^{-1}\theta R$ rewards from revealing their randomness, by Lemma 1. It follows that for a rational miner, this strategy can be advantageous to plain selfish mining only in case:

$$\frac{\alpha'}{1-\alpha'} > (1+\theta(1+\epsilon)^{-1}R)\frac{\alpha}{1-\alpha}$$

If we assume a miner can increase their effective hash rate by a factor of c, using low-entropy exponents, then their advantage in the low entropy case is $\alpha' = \alpha c/(\alpha c+\beta)$, where $\beta = 1-\alpha$ is the relative mining power of all other miners. If follows that the miner benefits if and only if:

$$\begin{aligned} \tfrac{\alpha c}{\alpha c+\beta} \cdot \tfrac{\alpha c+\beta}{\beta} &> (1+\theta(1+\epsilon)^{-1}R)\tfrac{\alpha}{\beta} \\ \iff c &> 1+\theta(1+\epsilon)^{-1}R \end{aligned}$$

If we adopt a sufficiently large intended time interval between blocks it is possible to bound the relative savings of a selfish miner using low-entropy exponents; following the parameterisation of subsection 3.4.2, if a selfish miner using such exponents can improve their hashing power by at most a multiplicative factor c then we can mitigate such attack by setting R to $(c-1)/(\theta(1+\epsilon)^{-1})$.

3.6 Discussion

While the clean generation of a new reference string from a ledger protocol is itself useful, real-world situations are likely to be more complex. In this section we discuss practical adjustments that may be made.

3.6.1 Upgrading Reference Strings

As distributed ledgers are typically long-lived, and may well outlive any reference string used within it – or have been running before a reference string was needed. Indeed, the Zcash protocol has seen upgrades in its reference string. A reference string being replaced with a new one is innocuous without further context, however, it is important to consider how they are usually used in zero-knowledge proofs. If the proof they are used in is stateless, upgrading from an insecure to a secure reference string behaves as one may naively expect: It ensures that after the upgrade, security properties hold.

In the example of Zcash, which runs a variant of the Zerocash [BCG+14] protocol, the situation is more muddy. Zerocash makes *stateful* zero-knowledge proofs. Suppose a user is sceptical of the security of the initial setup – and there is good reason to be [SWB19] – but is convinced the second reference string is secure. Is such a user able to use Zcash with confidence in its security?

Had Zcash not had safeguards in place, the answer would be no. While the protocol may operate as intended currently, and the user can be convinced of that, due to the stateful nature of the proofs, the user cannot be convinced of the correctness of this state. The Zcash cryptocurrency did employ similar safeguards to those we outline below. We stress the importance of such here, as not every project may have the same foresight.

Specifically, for a Zerocash-based system, an original reference string's backdoor could have been used to create mismatched transactions, and to effectively "mint" large coins illicitly. This process is undetectable at the time, and the minted coins would persist across a reference string upgrade. Our fictitious user may therefore be rightfully suspicious as to the value of any coins he is sold – they may be a part of an almost infinite pool!

Such an attack, once carried out (especially against a currency) is hard to recover from – it is impossible to identify "legitimate" owners of the currency, even if the private transaction history were deanonymised, and the culprit identified. The culprit may have traded whatever he created already. Simply invalidating the transaction would therefore harm those he traded with, not himself. In an extreme case, if he traded one-to-one with legitimate owners of the currency, he would succeed in effectively stealing the honest users funds. If such an attack is identified, the community has two unfortunate options: Annul the funds of potentially legitimate users, or accept a potentially large amount of inflation.

We may assume a less grim scenario however: Suppose we are *reasonably confident* in the security of our old reference string, but we are *more confident* of the new one. Is it possible to convince users that we have genuinely upgraded our security? We suggest the usage of a type of *firewalling* property. Such properties are common in the domain of cross-chain transfers [GKZ19], and are designed to prevent a catastrophic failure on one chain damaging another.

For monetary transfers, the firewall would guarantee an upper-bound of funds was not exceeded. Proving the firewall property is preserved is easy if a small loss of privacy is accepted – each private coin being re-minted before it can be used after the upgrade, during which time its value must be declared. Assuming everything operates fine, and the firewall property is not violated, users interacting with the post-firewall state can be confident as to the upper bound of funds available. Further, attacks on the system can be identified: If an attacker mints too many coins, eventually the firewall property will be violated, indicating that too many coins were in circulation – bringing the question of how to handle this situation with it. We believe that a firewall property does however give peace of mind to users of the system, and is a practical means to assuage concerns about the security of a system which once had a questionable reference string.

In Zcash, a soft form of such firewalling is available, in that funds are split across several "pools", each of which uses a different proving mechanism. The total value of each pool can be observed, and values under zero would be considered a cause for alarm, and rejected. Zcash use the terminology "turnstiles" [Zca19], and no attacks have been observed through them.

A further consideration for live systems is that as subsection 3.4.2 shows, the time required strongly

depends on the frequency between blocks. This may conflict with other considerations for selecting the block time – a potential solution for this is to only perform updates on "superblocks": blocks which meet a higher proof-of-work (or other selection mechanism) criteria than usual.

3.6.2 The Root of Trust

An important question for all protocols in the distributed ledger setting is whether a user entering the system at some point during its runtime can be convinced to trust in its security. Early proof-of-stake protocols, such as [KRDO17a], did poorly at this, and were subject to "stake-bleeding" attacks [GKR18] for instance – effectively meaning new users could not safely join the network.

For reference strings, if a newly joining user is prepared to accept that the honest majority assumption holds, they may trust the security of the reference string, as per Theorem 3. There is a curious difference to the security of the consensus protocol however: to trust the consensus – at least for proof-of-work based protocols – it is most important to trust a *current* honest majority, as these protocols are assumed to be able to recover from dishonest majorities at some point in their past. The security of the reference string on the other hand only relies on assuming honest majority during the initial δ time units. This may become an issue if a large period of time passes – why should someone trust the intentions of users during a different age?

In practice, it may make sense to "refresh" a reference string regularly to renew faith in it. It is tempting to instead continuously perform updates, however, as noted in subsection 3.6.1, this does not necessarily increase faith in a stateful system, although is can remove the "historical" part from the honest majority requirement when used with stateless proofs.

Most subversion attacks are detectable – they require lengthy forks which are unlikely to occur during a legitimate execution. In an optimistic case, where no attack is attempted, this may provide an additional level of confirmation: if there are no widespread claims of large forks during the initial setup, then the reference string is likely secure (barring large-scale out-of-band censorship). A flip side to this is that it may be a lot easier to sow doubt, however, as there is no way to *prove* this: A malicious actor could create a fork long after the initial setup, and claim that it is evidence of an attack to undermine the credibility of the system.

3.6.3 Applications to Non-Updateable SNARKs

Updateable SNARK schemes have two distinct advantages which our protocol makes use of: First, they have an explicit update procedure which allows a party ϕ to replace a reference string whose security depends on some assumption A, with one whose security depends on $A \vee (\phi \text{ is honest})$. Second, they can survive with a partially biased reference string, a fact which we don't use directly in this chapter, however, the functionality $\mathcal{F}_{\mathsf{uSRS}}$ we provide permits rejection sampling, encoding it into the ideal world.

The lack of an update algorithm can be resolved for some zk-SNARKs, such as [Gro16], by the existence of a weaker property: In two phases, the reference string can be constructed with (potentially different) parties performing round-robin updates (also group exponentiations) in each phase. This approach is also detailed in [BGM17], and it implies a natural translation to our protocol, in which the first phase is replaced with two phases of the same length, performing the first and second phase updates respectively.

The security of partially biased references strings has not been sufficiently analysed for non-updateable SNARKs, however, this weakness can be mitigated. Following [BGM17], it is possible to use a pure random beacon (as opposed to the resettable one used in section 3.5) to create a "pure" reference string from the "impure" one presented so far. To sketch the design: The random beacon would be queried after time δ, and the randomness used to select a trapdoor permutation over the reference string. This would then be applied by each party independently, arriving at the same – randomly distributed – reference string.

As this is not required for updateable SRS schemes, we did not perform this analysis in depth. However, the approach to the simulation would be to perform the SRS generation identically, and then program the random beacon to invert all permutations applied to the honest reference string. Since this includes the one honest permutation applied on every honest update, this is indistinguishable from a random value to the adversary. It is worth noting that the requirement of a random beacon is on the stronger side of requirements, especially as it should itself not allow adversarial influence to provide the desired advantage. Approaches using block hashes for randomness introduce exactly the limited influence which we are attempting to remove!

3.7 The Sonic uSRS

Sonic's uSRS [MBKM19, Section 4.3] consists of a series of exponentiations of group elements in pairing groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of prime order q, where a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ exists. Specifically, given generators $g \in \mathbb{G}_1, h \in \mathbb{G}_2$ and a depth parameter $d \in \mathbb{Z}_q$, the SRS has a trapdoor of $(\alpha, x) \in \mathbb{F}_q^{*2}$, with $\tau_0 = (1, 1)$.

The corresponding structure function is defined as:

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

3.7.1 Specification of Sonic Updates

We omit the $e(g, h^\alpha)$ term presented in Sonic, as this can be computed from the rest of the SRS, and is therefore immaterial to the update procedure. The permitted trapdoor permutations are field multiplications:

$$P := \{\ (\alpha, x) \mapsto (\alpha\beta, xy) \mid (\beta, y) \in \mathbb{F}_q^{*2}\ \}.$$

Correspondingly, † exponentiates group elements:

$$\begin{aligned} p = (\alpha, x) \mapsto (\alpha\beta, xy) \implies \\ p^\dagger = \left(\{G_i, H_i, H'_i\}_{i=-d}^{d}, \{G'_i\}_{i=-d, i\neq 0}^{d} \right) \\ \mapsto \left(\left\{ G_i^{y^i}, H_i^{y^i}, H_i'^{\beta y^i} \right\}_{i=-d}^{d}, \left\{ G_i'^{\beta y^i} \right\}_{i=-d, i\neq 0}^{d} \right) \end{aligned}$$

Observe that field *multiplications* over α or x can efficiently be applied to the corresponding structure through exponentiation: $g^{(\alpha x^i)\beta y^i} = (g^{\alpha x^i})^{\beta y^i}$. The full update proof procedure is as follows:

procedure ProveUpd(srs, p)
  let $(\beta, y) \leftarrow p((1, 1))$
  return $(g^y, g^{\beta y}, \pi)$

The verification procedure ensures correct computation by checking the consistency of various pairing computations:

procedure VerifyUpd(srs, ρ, srs′)
  let $(\{G_i, H_i, H'_i\}_{i=-d}^{d}, \{G'_i\}_{i=-d, i\neq 0}^{d}) \leftarrow$ srs
  let $(\{I_i, J_i, J'_i\}_{i=-d}^{d}, \{I'_i\}_{i=-d, i\neq 0}) \leftarrow$ srs′
  let $(A, B, \pi) \leftarrow \rho$
  if
    $e(I'_1, h) \neq e(B, H'_1) \vee e(g, J'_1) \neq e(B, H'_1) \vee e(I_1, h) \neq e(A, H_1) \vee e(g, J_1) \neq$
    $e(A, H_1) \vee I_0 \neq g \vee J_0 \neq h$ **then**
    return 0
  for $i = -d$ **to** d **do**
    if
      $\neg(i = d \vee e(I_i, J_1) = e(I_1, J_i) = e(I_{i+1}, h) =$
      $e(g, J_{i+1})) \vee \neg(e(I_i, J'_0) = e(g, J'_i)) \vee$
      $(i \neq 0 \wedge \neg e(I_i, J'_0) = e(I'_i, h))$ **then**

return 0
return 1

3.7.2 Satisfaction of Security Properties

Theorem 4. *Sonic, as described in subsection 3.7.1, is an updatable reference string scheme, satisfying correctness, structure preservation, update uniformity, update extraction, permutation extraction, and permutation lifting.*

Proof. We prove each property individually.

Correctness Follows from all pairing checks being satisfied. □

Structure preservation Suppose a structured input $S(\tau)$, and an update proof ρ, and a new SRS srs', where:

$$S(\tau) = \left(\left\{g^{x^i}, h^{x^i}, h^{\alpha x^i}\right\}_{i=-d}^{d}, \left\{g^{\alpha x^i}\right\}_{i=-d, i\neq 0}^{d}\right)$$
$$\mathsf{srs}' = \left(\left\{g^{k_i}, h^{m_i}, h^{n_i}\right\}_{i=-d}^{d}, \left\{g^{l_i}\right\}_{i=-d, i\neq 0}^{d}\right)$$
$$\rho = (g^y, g^{\beta y})$$

If VerifySRS returns 1, we know all of the following hold, due to the conditions checked:

- $e(g^{l_1}, h) = e(g, h^{n_1}) = e(g^{\beta y}, h^{\alpha x})$
- $e(g^{k_1}, h) = e(g, h^{m_1}) = e(g^y, h^x)$
- $\forall i \in [-d, d) : e(g^{k_i}, h^{m_1}) = e(g^{k_1}, h^{m_i}) = e(g^{k_{i+1}}, h) = e(g, h^{m_{i+1}})$
- $\forall i \in [-d, d] : e(g^{k_i}, h^{n_0}) = e(g, h^{n_i})$
- $\forall i \in [-d, d] \setminus \{0\} : e(g^{k_i}, h^{n_0}) = e(g^{l_i}, h)$

As $e(g, h)$ is a generator over $\mathbb{G}_T$, and each of the above can be expressed as an equality of exponentiations of the form $e(g, h)^a = e(g, h)^b$, we simplify these to equalities within $\mathbb{F}_q^*$ of their exponents:

- $l_1 = n_1 = \alpha\beta xy$
- $k_1 = m_1 = xy$
- $\forall i \in [-d, d) : k_i m_1 = k_1 m_i = k_{i+1} = m_{i+1}$
- $\forall i \in [-d, d] : k_i n_0 = n_i$
- $\forall i \in [-d, d] \setminus \{0\} : k_i n_0 = l_i$

It follows directly that $n_0 = \alpha\beta$, $k_i = m_i = (xy)^i$, and $l_i = n_i = \alpha\beta(xy)^i$. As a result, srs' matches exactly the structured reference string $S((\alpha\beta, xy)) = p^\dagger(S(\tau))$. □

Update uniformity Let $\tau = (\alpha, x)$. $p \xleftarrow{R} P$ is defined by a multiplication with two uniformly sampled field elements in $\beta, y \xleftarrow{R} \mathbb{F}_q^*$, such that the trapdoor $p(\tau) = (\alpha\beta, xy)$. Due to multiplication in prime fields with a fixed element (here α and x) being a bijective function, the result $(\alpha\beta, xy)$ is also distributed uniformly at random in $\mathbb{F}_q^{*2}$, therefore being indistinguishable from a new, randomly sampled trapdoor. □

Update proof simulation We present the following simulation algorithm:

procedure $\mathcal{S}_\rho((\alpha, x), \mathsf{srs})$
$(\{G_i, H_i, H'_i\}_{i=-d}^{d}, \{G'_i\}_{i=-d, i\neq 0}^{d}) \leftarrow \mathsf{srs}$
return $\left(G_1^{(x^{-1})}, G_1'^{(x^{-1})(\alpha^{-1})}\right)$

This utilises only a small number of efficient group operations, and is therefore PPT. As the VerifyUpd pairing checks all succeed, the returned update proof will verify. □

Permutation Extraction Observe that

$$\mathcal{R}((A, B), p) \iff \text{let } (a, b) = p((1, 1)) \text{ in } A = g^a \wedge B = g^b.$$

A straightforward encoding of p is the pair of field elements (a, b). This relation is clearly in NP, and is also a bijection due to the relation of $\mathbb{G}_1$ and $\mathbb{F}_q^*$. □ □

3.7.3 Instantiating $\mathcal{F}_{\mathsf{NIZK}}$

We can employ Fischlin's transform [Fis05] in combination with a simple sigma protocol to prove knowledge of pairs of exponents. Specifically, we propose the parallel composition of two Schnorr proofs of knowledge of exponent [Sch90b]. It is important to treat these as a single proof, and not two separate proofs, as the latter would enable the adversary to create proofs which are only partially extractable. We posit that these would still allow for simulation, however, the simulator would be tasked with a more difficult, and implementation specific book-keeping.

3.8 The Nakamoto Ledger

The basic functionality of this ledger allows the submission of transactions, and retrieving each of the following:

- A confirmed prefix of the ledger state.
- A "projection" of the ledger state – i.e. what the local state will approach, if there is no chain reorganisation.
- The confirmed "leader state", which models the mechanism used for the SRS generation.

When any of these values is queried, the functionality ensures that liveness and chain quality properties still hold. The adversary further has the power to instruct the creation of a new block on behalf of any party, and to instruct any party to adopt a different chain. In both cases, the functionality ensures that the common prefix property is preserved. The adversary has full control over the contents of both honest and adversarial blocks, as well as their order.

3.8.1 Functionality Definition

Functionality $\mathcal{F}_{\mathsf{nakLedger}}$

A ledger following a Nakamoto-style consensus, with each party having a *projected* chain, a prefix of which is common to all parties. Common prefix, chain quality and chain growth are guaranteed.

State variables and initialisation values:

Variable	Description
$\Pi := \phi \mapsto \epsilon$	Mapping of parties to projected ledger states
$T := \varnothing$	Multiset of submitted transactions
$\mathsf{hon} := \varnothing$	Mapping of block ids to 1 if they are honest, or 0 if not

When receiving a message $(\textsc{submit}, \mathsf{tx})$ *from a party* ϕ*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
let $T \leftarrow T \cup \{(\mathsf{tx}, t)\}$
query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathsf{tx}, t)$

When receiving a message READ *from a party* ϕ*:*

assert $\mathsf{liveness}(\phi) \wedge \mathsf{chainQuality}(\phi)$
return $\mathsf{map}(\mathsf{proj}_1, \mathsf{txs}(\Pi(\phi)^{\lceil k}))$

When receiving a message PROJECTION *from a party* ϕ*:*

assert $\mathsf{liveness}(\phi) \wedge \mathsf{chainQuality}(\phi)$
return $\mathsf{map}(\mathsf{proj}_1, \mathsf{txs}(\Pi(\phi)))$

When receiving a message LEADER-STATE *from a party* ϕ*:*

assert $\mathsf{liveness}(\phi) \wedge \mathsf{chainQuality}(\phi)$
let $\vec{a} \leftarrow \mathsf{map}(\lambda(\cdot, a, \cdot, t) : (a, t), \Pi(\phi)^{\lceil k})$
return $\mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$

When receiving a message $(\textsc{extend}, \phi, B, t, a)$ *from* $\mathcal{A}$*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t'
let $\mathsf{id} \xleftarrow{R} \{0,1\}^\kappa$
if $\phi \in \mathcal{H}$ **then**
  let $\vec{a} \leftarrow \mathsf{map}(\lambda(\cdot, a, \cdot, t) : (a, t), \Pi(\phi))$
  let $\sigma \leftarrow \mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$
  let $a \xleftarrow{R} \mathsf{Gen}(\sigma, t')$
  let $t \leftarrow t'$ **let** $\mathsf{hon}(\mathsf{id}) \leftarrow 1$
else
  let $\mathsf{hon}(\mathsf{id}) \leftarrow 0$
  if $t' < t$ **then let** $t \leftarrow t'$
  else if $\exists t'' : (\cdot, \cdot, \cdot, t'') = \mathsf{last}(\Pi(\phi)) \wedge t'' > t$ **then let** $t \leftarrow t''$
let $\Pi(\phi) \leftarrow \Pi(\phi) \parallel (B, a, \mathsf{id}, t)$
assert $\forall \phi' \in \mathcal{P} : \Pi(\phi)^{\lceil k} \prec \Pi(\phi')$
return (B, a, id, t)

When receiving a message $(\textsc{advance}, \phi, \Sigma')$ *from* $\mathcal{A}$*:*

assert $\exists \phi' \in \mathcal{P} : \Sigma' \prec \Pi(\phi')$
assert $\forall \phi' \in \mathcal{P} : \Sigma'^{\lceil k} \prec \Pi(\phi') \wedge \Pi(\phi')^{\lceil k} \prec \Sigma'$
let $\Pi(\phi) \leftarrow \Sigma'$

Helper procedures:

function $\mathsf{txs}(\Pi_\phi)$
  let $\vec{B} \leftarrow \mathsf{map}(\mathsf{proj}_1, \Pi_\phi)$
  return $\mathsf{concat}(\vec{B})$

procedure $\mathsf{liveness}(\phi)$
  send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
  if
    $\exists t_0 < t : |[\ t_b \mid (\cdot, \cdot, \cdot, t_b) \in \Pi(\phi), t_0 - s \leq t_b < t_0\ ]|$
    $< \gamma \wedge t_0 - s \geq 0$ **then**
    return $\perp$
  return $\forall (\mathsf{tx}, t') \in T : t' + \lceil (l+k)\gamma^{-1} \rceil s > t \ \vee (\mathsf{tx}, t') \in \mathsf{txs}(\Pi(\phi)^{\lceil k})$

procedure $\mathsf{chainQuality}(\phi)$
  let $\vec{\mathsf{id}} \leftarrow \mathsf{map}(\mathsf{proj}_3, \Pi(\phi)^{\lceil k})$
  return $\forall i \in \mathbb{Z}_{|\vec{a}|-l} : \left(\sum_{j \in \mathbb{Z}_l} \mathsf{ids}(\vec{\mathsf{id}}_{i+j})\right) \geq \mu l$

In order to judge chain growth, this functionality needs access to a simple global clock, given in subsection 3.11.1.

3.8.2 Relation to Existing Protocols

Existing UC treatments of Nakamoto-style ledgers, such as [BGK+18, BMTZ17] already provide functionalities which provide persistence and liveness guarantees. Moreover, the protocols used in their implementation have been independently shown to satisfy the properties of common prefix, chain quality, and chain growth.

Given similar assumptions, such as a limited random oracle, and a synchronous or semi-synchronous network, these protocols will also fit the $\mathcal{F}_{\mathsf{nakLedger}}$ functionality presented above. Notably the UC-proof of these ledgers relies on first proving these three chain properties, then proving persistence and liveness, and finally concluding that these satisfy their ledger functionality.

$\mathcal{F}_{\mathsf{nakLedger}}$ exposes more of the internals of the protocol – the fact that there is a chain selection process, and that this is subject to the constraints of common prefix, chain quality, and chain growth – but otherwise does not greatly change the ideal world behaviour.

Due to this strengthened functionality being designed to merely expose more of the well-understood protocol properties, we conjecture that UC implementations which have a proof relying on the three Nakamoto chain properties can be used to realise $\mathcal{F}_{\mathsf{nakLedger}}$, with a large part of the proof applying directly.

3.9 The Adaptor Protocol

We provide a small protocol which adapts the honest interface of the Nakamoto ledger to match that of the ideal world – specifically ensuring the leadership state seen matches the ideal world's, and that the SRS is read only if sufficient time has passed.

Protocol LEDGER-ADAPTOR

The protocol adaptor fits the interface of $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ to match those of $\mathcal{F}_{\mathsf{uSRS}}$ and $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$. It operates in the $(\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}, \mathcal{G}_{\mathsf{clock}})$-hybrid world.

When receiving a message $(\textsc{submit}, \mathsf{tx})$ *from a party* ϕ*:*
  send $(\textsc{submit}, \mathsf{tx})$ **to** $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$
When receiving a message READ *from a party* ϕ*:*
  send READ **to** $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ and **receive the reply** txs
  return txs
When receiving a message PROJECTION *from a party* ϕ*:*
  send PROJECTION **to** $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ and **receive the reply** txs
  return txs
When receiving a message LEADER-STATE *from a party* ϕ*:*
  send LEADER-STATE **to** $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ and **receive the reply** $(\cdot, \sigma^{\mathsf{ideal}})$
  return σ^{ideal}
When receiving a message SRS *from a party* ϕ*:*
  send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

if $t < \delta$ **then return** $\perp$
else

send LEADER-STATE **to** $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ and
receive the reply $(\mathsf{srs}, \cdot)$
return srs

Forward SUBMIT, READ, *and* PROJECTION *queries to* $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$

3.10 The Simulator

Simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$

The simulator between the protocol adaptor over $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$, and $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$ and $\mathcal{F}_{\mathsf{uSRS}}$. It operates in the $\mathcal{G}_{\mathsf{clock}}$-hybrid world.

State variables and initialisation values:

Variable	Description
$\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$	A simulation of the hybrid-world ledger
$\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$	A simulation of the low-level NIZK functionality
$A := \varnothing$	Map from honest updates to the applied permutation

When receiving a message $(\textit{TRANSACTION}, \mathsf{tx}, t)$ *from* $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$*:*
simulate sending $(\text{SUBMIT}, \mathsf{tx})$ **to** $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$
When receiving a message $(\textit{SUBMIT}, \mathsf{tx})$ *from* $\mathcal{A}$ *for* $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$*:*
send $(\text{SUBMIT}, \mathsf{tx})$ **to** $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$
When receiving a message $(\textit{PERMUTE}, \phi)$ *from* $\mathcal{F}_{\mathsf{uSRS}}$*:*

simulate sending LEADER-STATE **to** $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$
through ϕ and
receive the reply $(\mathsf{srs}, \cdot)$
let $\vec{a} \leftarrow \mathsf{map}(\mathsf{proj}_2, \mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi))$
return $\mathcal{X}_p(\vec{a})$
When receiving a message $(\textit{EXTEND}, \phi, B, t, a)$ *from* $\mathcal{A}$ *for* $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$*:*
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t'
if $\phi \in \mathcal{H} \wedge t' \leq \lceil l\gamma^{-1} \rceil s$ **then**
let $\vec{a} \leftarrow \mathsf{map}(\mathsf{proj}_2, \mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi))$
let $(\mathsf{srs}, \cdot) \leftarrow \mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$
let $p \leftarrow \mathcal{X}_p(\vec{a})$
if $p^{-1\dagger}(\mathsf{srs}) \neq S(\tau_0)$ **then**
// We cannot extract a trapdoor;
// the SRS is already secure
let $p' \xleftarrow{R} P; \rho \leftarrow \mathsf{ProveUpd}(\mathsf{srs}, p')$
let $\mathsf{srs}' \leftarrow p'^{\dagger}(\mathsf{srs})$

simulate sending (PROVE, ρ, p') **to** $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ and
receive the reply π
else
// We produce an update to match a
// random "initial" SRS
let $\tau \leftarrow p(\tau_0)$
let $p' \xleftarrow{R} P$

send HONEST-SRS **to** $\mathcal{F}_{\mathsf{uSRS}}$ and
receive the reply $\mathsf{srs}_{\mathcal{H}}$
let $\mathsf{srs}' \leftarrow p'^{\dagger}(\mathsf{srs}_{\mathcal{H}})$
let $\rho \leftarrow \mathcal{S}_\rho(p(\tau), \mathsf{srs}')$

query $\mathcal{A}$ **with** (PROVE, ρ) and **receive the reply** π,
satisfying $\pi \neq \bot \wedge (\rho, \pi) \notin \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\overline{\Pi} \wedge (\cdot, \pi) \notin \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi$, else **sampling from** $\{0,1\}^\kappa$
let $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi \leftarrow \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi \cup \{(\rho, \pi)\}$
let $A(\rho) \leftarrow p$
let $a^{\mathsf{ideal}} \leftarrow \bot$
else if $\phi \in \mathcal{H}$ **then**
let $\mathsf{srs}', \rho, \pi \leftarrow \epsilon$
let $a^{\mathsf{ideal}} \leftarrow \bot$
else let $(\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}) \leftarrow a$

send (EXTEND, $\phi, B, t, a^{\mathsf{ideal}}$) **to** $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$ and
receive the reply $(B, a^{\mathsf{ideal}}, \mathsf{id}, t)$
if $\phi \in \mathcal{H}$ **then**
let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\mathsf{hon}(\mathsf{id}) \leftarrow 1$
else
let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\mathsf{hon}(\mathsf{id}) \leftarrow 0$

let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi) \leftarrow \mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi) \,\|\, (B, (\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}), \mathsf{id}, t)$

assert $\forall \phi' \in \mathcal{P} : \mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi)^{\lceil k} \prec \mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}.\Pi(\phi')$
return $(B, (\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}), \mathsf{id}, t)$

When receiving a message (ADVANCE, ϕ, Σ') *from* $\mathcal{A}$ *for* $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$*:*
simulate sending (ADVANCE, ϕ, Σ') **to** $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$
// Remove SRS updates from Σ'
let $\Sigma' \leftarrow \mathsf{map}(\lambda(B, (\cdot, \cdot, \cdot, a^{\mathsf{ideal}}), t) : (B, a^{\mathsf{ideal}}, t), \Sigma)$
send (ADVANCE, ϕ, Σ') **to** $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$

Forward requests to $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$*, and all other adversarial messages for* $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ *to* $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$*.*

Helper procedures:
procedure $\mathcal{X}_p(\vec{a})$
let $p \leftarrow \mathsf{id}$
let $\mathsf{srs} = S(\tau_0)$
for $(\mathsf{srs}', \rho, \pi, \cdot)$ **in** $\vec{a}$ **do**
// Skip invalid updates
if $\neg\mathsf{VerifyUpd}(\mathsf{srs}, \rho, \mathsf{srs}') \vee (\rho, \pi) \notin \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi$ **then continue**
let $\mathsf{srs} \leftarrow \mathsf{srs}'$
if $(\rho, \pi) \in \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W$ **then**
let $p \leftarrow \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W((\rho, \pi)) \circ p$
else if $\rho \in A$ **then**
// The update is honest.
// Start with its permutation.
let $p \leftarrow A(\rho)$
else
// A witness-less adversarial update
// was encountered.
abort
return p

3.11 Minor UC Functionalities

3.11.1 The Global Clock

Functionality $\mathcal{G}_{\mathsf{clock}}$

The global clock allows parties to agree on some discrete notion of time.

State variables and initialisation values:

Variable	Description
$t := 0$	Current time
$T := \varnothing$	Timekeepers
$A := \varnothing$	Agreements to advance

When receiving a message REGISTER *from a party ϕ:*
  let $T \leftarrow T \cup \{\phi\}$
When receiving a message DEREGISTER *from a party ϕ:*
  let $T \leftarrow T \setminus \{\phi\}$
When receiving a message UPDATE *from a party ϕ:*
  let $A(\phi) \leftarrow \top$
  if $\forall \phi \in T : A(\phi)$ **then**
    let $t \leftarrow t + 1; A \leftarrow \lambda\phi : \bot$
    query $\mathcal{A}$ **with** TICK-TOCK
When receiving a message READ *from a party ϕ:*
  return t

3.11.2 Non-Interactive Zero-Knowledge

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

The (malleable) non-interactive zero-knowledge functionality $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ allows proving of statements in an NP relation $\mathcal{R}$.

State variables and initialisation values:

Variable	Description
$W := \varnothing$	Mapping of statement/proof pairs to witnesses
$\Pi := \varnothing$	Set of statement/proof pairs
$\overline{\Pi} := \varnothing$	Set of known invalid statement/proof pairs

When receiving a message $(\textsc{prove}, x, w)$ *from a party ϕ:*
  if $\neg x \mathcal{R} w$ **then**
    return $\bot$
  query $\mathcal{A}$ **with** $(\textsc{prove}, x)$ and **receive the reply** π,
    satisfying $\pi \neq \bot \wedge (x, \pi) \notin \overline{\Pi} \wedge (\cdot, \pi) \notin \Pi$, else **sampling from** $\{0,1\}^\kappa$
  let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}; W(x, \pi) \leftarrow w$
  return π
When receiving a message $(\textsc{maul}, x, \pi, \pi')$ *from $\mathcal{A}$:*
  if $(x, \pi) \in \Pi$ **then**
    let $\Pi \leftarrow \Pi \cup \{(x, \pi')\}$
    let $W(x, \pi') \leftarrow W(x, \pi)$
When receiving a message $(\textsc{verify}, x, \pi)$ *from a party ϕ:*
  if $(x, \pi) \notin \Pi \cup \overline{\Pi} \wedge \pi \neq \bot$ **then**

query $\mathcal{A}$ **with** (VERIFY, x, π)
// The adversary has been given a chance to
// prove the statement. It didn't take it.
if $(x, \pi) \notin \Pi \cup \overline{\Pi}$ **then**
let $\overline{\Pi} \leftarrow \overline{\Pi} \cup (x, \pi)$
return $(x, \pi) \in \Pi$

3.11.3 Random Oracle

Functionality $\mathcal{F}_{\mathsf{RO}}$

The random oracle functionality $\mathcal{F}_{\mathsf{RO}}$ returns a uniform random value in $\{0,1\}^\kappa$ for each input.

State variables and initialisation values:

Variable	Description
$H := \varnothing$	A map from inputs to (fixed) outputs

When receiving a message (QUERY, x) *from a party* ϕ*:*

if $x \notin H$ **then let** $H(x) \xleftarrow{R} \{0,1\}^\kappa$
return $H(x)$

3.11.4 Delay Wrapper

Functionality $\mathcal{W}_{\mathsf{delay}}(\delta, \mathcal{F})$

The wrapper functionality $\mathcal{W}_{\mathsf{delay}}(\delta, \mathcal{F})$ of $\mathcal{F}$ accepts *honest* inputs only after δ time slots.

When receiving a message M *from a party* ϕ*:*
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
if $t < \delta \wedge \phi \in \mathcal{H}$ **then return** $\perp$
else
send M **to** $\mathcal{F}$ and **receive the reply** y
return y

3.12 Security Analysis

As for any UC proof, we require a simulator which ensures the ideal world behaves indistinguishably from the real world. Our simulator, $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$, is formally described in section 3.10. Intuitively, this simulator ensures that the real and ideal world's ledgers are equivalent, and that the real world uSRS is equal to the uSRS produced in the ideal world.

In order to achieve this, the simulator ensures that the initial honest reference string provided by $\mathcal{F}_{\mathsf{uSRS}}$ is the basis of the uSRS of a simulated execution of the real-world protocol. Doing so relies primarily on three things: First, the simulator's ability to extract the permutation from any adversarial reference string update. Second, the simulator's ability to, given the adversarial trapdoors, then produce a valid "honest" update which ensures the reference string is a random permutation of the ideal-world honest string $S(\tau_{\mathcal{H}})$. And finally, the simulator's knowledge that the final reference string in its simulation will have at least one honest update.

The simulator observes each of the competing chains, and when the first honest update occurs in each, coerces the simulated update into a permutation of the ideal honest reference string. For each

subsequent honest update, the simulator performs the update normally, remembering the randomness used. Combined with extracting from adversarial updates, the simulator either knows the *entire* trapdoor of the reference string (if there was no honest update), or all except for the first honest update. By the backbone properties enforced by $\mathcal{F}_{\mathsf{nakLedger}}$, the simulator knows that the first case will not apply, and that only one prefix of valid updates will exist, after δ time has passed. As a result, the simulator knows exactly which permutation to apply to the honest ideal reference string to match the real world's result.

As $\mathcal{F}_{\mathsf{uSRS}}$ only provides a single honest SRS, the simulator applies a random permutation to this for each initial honest update, ensuring that the updates of different chains remain unlinkable.

We will prove UC-emulation, and will therefore refer to the ideal and real worlds frequently throughout the proof. Beyond this, the simulator locally simulates the NIZK functionality and the ledger functionality. To be clear which functionality we are talking about at any point, we will use $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$, $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$, and $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$ to refer to the ideal, simulated, and real ledgers respectively. We refer to the real-world NIZK functionality as $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, and the simulted NIZK as $\mathcal{S}_{\text{LEDGER-ADAPTOR}}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$. The notation $\mathcal{F}.x$ is used to mean "the variable x within the functionality $\mathcal{F}$" – it is also used to refer to the ideal trapdoor $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$.

Our simulator, which we assume is provided with the update simulation algorithm $\mathcal{S}_\rho$, and can extract permutations from adversarial updates via a simulated NIZK, is equipped with a helper function $\mathcal{X}_p$. Given a series of updates, $\mathcal{X}_p$ computes the permutation applied to the reference string's trapdoor as far back as possible. It receives as inputs the sequence of updates $\vec{a}$, and has access to a mapping W from NIZK statements and proofs to corresponding witnesses (as far as the simulator knows them), and a mapping A from honest updates to the permutation applied to the honest SRS. It returns a permutation in P, which can be applied either to the initial trapdoor τ_0, or the initial honest trapdoor $\tau_{\mathcal{H}}$, to create the same SRS as the sequence of updates. We prove this in the following auxiliary lemma that will be used in the proof of our main theorem.

Lemma 2. *In the ideal-world execution of $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$, $\mathcal{X}_p(\vec{a})$ outputs a permutation $p \in P$, such that its inverse, applied to the underlying trapdoor of the SRS generated from the given sequence of updates $\vec{a}$, is either the initial trapdoor τ_0, or the honest trapdoor $\tau_{\mathcal{H}}$.*

Proof. The output of $\mathcal{X}_p$ is either id, a permutation in the mapping A, a permutation recorded by the simulated NIZK, or a series of function compositions of the above. As only permutations in P are stored in A, $\mathsf{id} \in P$, and as P is closed under composition, the returned permutation is in P. The permutation applied corresponds directly to how the underlying trapdoor of the uSRS is updated by longest suffix of updates in $\vec{a}$ for which the trapdoor is known – i.e. the trapdoor permutation is recorded in $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W$, or a permutation of the honest trapdoor is recorded in A. When this isn't the case, the update is skipped, and the trapdoor reset, ensuring that any trapdoors preceeding a non-extractable value are ignored. The case that the trapdoors are known for *all* of the updates is trivial; as by definition inverting this permutation will result in the initial trapdoor τ_0.

If, however, at any point the trapdoor is not recorded in $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W$ (despite $\mathsf{VerifyUpd}$ succeeding), at this point the trapdoor must be honestly generated: As this update was not skipped, the NIZK proofs associated with it must verify. The only way for the proofs to verify, and the NIZK functionality *not* to have recorded the corresponding witnesses, however, is that the simulator added the proof manually to the NIZK's set of valid proofs. This only happens at one point – when creating simulated NIZK proofs to accompany simulated update proofs, which is used *only* for random permutations applied to the honest reference string. While (if the adversary is capable of inverting the structure function) multiple honest updates may exist in the same chain, if at least one of them is a replayed update, the last such effectively "resets" the reference string to a known permutation of the honest reference string.

Finally, we note that for this witness-less update, the remaining trapdoor defines a permutation of $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. Algorithm $\mathcal{X}_p$ extracts the trapdoors from all subsequent updates to compute the permutation

applied to this honest trapdoor – ensuring precisely that inverting this permutation results in $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. □ □

of Theorem 3. If the environment can distinguish between these worlds, there must exist a minimal series of interactions the environment, combined with its adversary, can make to cause the other UC ITMs to behave sufficiently differently to allow distinguishing. We will show that for any interaction the environment makes, it will not learn enough information to distinguish the two worlds, and therefore that across *all* (polynomially many) interactions it also cannot distinguish. First, we consider what actions the adversary/environment pair can take. The interactions fall into the following categories:

1. Honest or adversarial SUBMIT, READ, LEADER-STATE, or PROJECTION queries
2. Interactions with $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, or $\mathcal{G}_{\mathsf{clock}}$
3. ADVANCE queries
4. EXTEND queries
5. SRS queries

We will establish the following invariants throughout the execution of the UC security game:

- $\mathcal{G}_{\mathsf{clock}}$ has the same internal state in both the real and ideal worlds.
- $\mathcal{S}_{\text{LEDGER-ADAPTOR}}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ has the same internal state as the real-world $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, except that it does not know the witnesses for honestly generated proofs or their mauled variants.
- $\mathcal{S}_{\text{LEDGER-ADAPTOR}}.\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$ has the same internal state as the real-world ledger $\mathcal{F}^{\mathsf{real}}_{\mathsf{nakLedger}}$, and differs from the ideal-world ledger $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$ only in that all state updates contain an addition SRS update term.

Ledger reads and submissions Given these invariants, it is clear that the environment cannot distinguish given the results of READ and PROJECTION queries – they must return the same value! Further, as the adaptor protocol strips the SRS component from the leader state, and the ideal world's leader state is precisely defined as being without this component, it is clear that also LEADER-STATE queries will be indistinguishable (even if made directly by the adversary, since these are answered by $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$). For SUBMIT queries by either the environment or the adversary, both worlds will add the transaction, with the current timestamp, to their ledger's submitted transactions, and will notify the adversary *once*, and return the transaction together with the timestamp. This does not reveal any information to the environment which could be used to distinguish.

Queries to other functionalities Likewise, $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ queries clearly will not permit the environment to distinguish, or invalidate the above mentioned invariants – they do not go beyond the NIZK functionality, and this does not read – only update – the witness map. Similarly for $\mathcal{G}_{\mathsf{clock}}$, as this exists in both worlds, and is not manipulated by the simulator (or any other entity), beyond read-only operations, it will behave identically.

advance queries The simulator first simulates advancing a specified party's ledger state on $\mathcal{F}^{\mathsf{simul}}_{\mathsf{nakLedger}}$. If this succeeds, the simulator knows that the advancement will succeed in the ideal world as well, where the ledger state is less constrained. It removes the SRS updates from the ledger state being switched to, and issues a corresponding advance query to $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$. If the simulated ADVANCE does not succeed, it will also have failed in the real world execution, both of which will abort. If the update succeeds, the invariant between the various ledger states is preserved – up to the lack of SRS updates in the ideal world, they are the same. If the update fails, both worlds terminate execution.

(extend, ϕ, B, t, a) queries Let us first detail the function of EXTEND queries. Called by the adversary, if the party parameter ϕ represents an honest party, the query runs Gen to generate a new update a to apply to this party's view of the leadership state. If the party is adversarial on the other hand, an adversary-supplied update parameter a is used instead. With the timestamp t (or the accurate time for honestly created blocks), block content B, state update a, and a randomly sampled ID, a new block is created, and appended to ϕ's projected chain. Finally, it is asserted that the common prefix property still holds.

Once the simulator intercepts such a query, it needs to ensure not only that the same EXTENDs are carried out in the simulated and ideal ledgers, but also that honest SRS updates are (when necessary) sourced from the $\mathcal{F}_{\mathsf{uSRS}}$ functionality. In the case that the party extending the chain is adversarial, this is simple – split the adversarial real world update a into an SRS update and an ideal-world update (it is worth noting that these need not be valid), and forward only the ideal-world update in an EXTEND query to $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{nakLedger}}$. This already results in the real and ideal ledgers satisfying the invariant, leaving the simulated ledger. For this, the simulator manually inserts the ID returned from the ideal-world ledger, inserts the new block, and asserts the same common prefix condition as the real world does, ensuring these two ledgers are in the same state and – crucially – abort under the same conditions. The returned value is identical to that returned in the real world.

For honest updates, things are more complex. If the current time is after when honest SRS updates are performed, the honest SRS update is set to ϵ, as in the real world. Otherwise, the SRS is reconstructed from the party's current projected ledger view, and the simulator attempts to extract the trapdoor permutation from this SRS. If it succeeds in extracting the entire trapdoor, the simulator ensures it is updated such that it can no longer do so: It updates the uSRS to a permutation of the honest uSRS $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$, by first applying a fresh permutation to it, recording this in the map A, and creating the corresponding update proof using $\mathcal{S}_\rho$.

By the update uniformity property, this is indistinguishable from the result of Gen, which the environment expects. In case the full trapdoor cannot be extracted, Gen is used to generate the "honest" SRS update, ensuring the simulator knows the trapdoor for this update as well (as it retains the NIZK witness used). Finally, the ideal ledger is sent an EXTEND query, with a^{ideal} set to $\perp$. Execution proceeds as in the adversarial case, with the SRS part of the update being distributed equally in the real and simulated ledgers, and the ideal-world component being generated directly by the ideal world functionality (and therefore also being distributed the same as in the real world, which samples from the same distribution).

srs queries Finally, a user may query the SRS. If this happens before time δ, both worlds return $\perp$ – the delay wrapper does so in the ideal world, and the adaptor protocol does so in the real world. Otherwise, the real world reconstructs the leadership state, and returns only the SRS component, while the ideal world queries the simulator for a trapdoor permutation, and, if the SRS is not yet finalised, applies it to the honest SRS.

Recall that after every extension, $\mathcal{F}_{\mathsf{nakLedger}}$ ensures that the common prefix property holds. Further, once a party's projected ledger state has some common prefix, this is only ever extended – either by extending the whole projection (in EXTEND), or by switching to a different one with the same prefix (in ADVANCE). After time δ, if chain quality and liveness hold, we can split each party's projected chain into two parts: Blocks with a timestamp at or before the time δ_1, and those with a timestamp after it. As EXTEND enforces timestamps to be monotonically increasing, these concatenate to form the entire chain. By the chain growth property, and as it is at least time δ, we know that the first part contains at least l blocks, and the second at least k blocks. Chain quality ensures that the first part contains at least μ honest blocks, while Apply ignores updates with a timestamps after δ_1. Combined, these facts imply that, for any party, the valid SRS updates, taken from their stable chain, are identical.

After the first SRS query, both the ideal and real worlds will not change what value they return,

the former because it has then recorded the final trapdoor, and the latter because the common prefix containing valid reference string updates cannot change. The first query is therefore the most interesting.

From Lemma 2, we know that the permutation p extracted by the simulator when it is queried for the SRS permutation will, inverted and applied to the SRS' underlying trapdoor, either result in τ_0, or $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. From the above we know that the SRS the simulator is extracting from matches that which honest parties generate – containing at least one honest update (by chain quality). As the first honest update in any chain is extracted from a $\mathcal{F}_{\mathsf{uSRS}}$-provided reference string, (and, by the correctness property, it *is* valid) it cannot be τ_0. Therefore, the simulator, by providing p to $\mathcal{F}_{\mathsf{uSRS}}$, satisfies its requirements of a permissible permutation in P, and ensures that once the permutation is applied, the same SRS is returned: $S(p(\tau_{\mathcal{H}})) = S(p(p^{-1}(\tau))) = S(\tau)$.

In the above we have brushed aside the issue of aborts, however, these are also simple to deal with. $\mathcal{F}_{\mathsf{uSRS}}$ aborts if given an invalid permutation, which the simulator does not do. In the real world, if liveness or chain quality are violated, $\mathcal{F}_{\mathsf{nakLedger}}$ aborts. In each query, the simulator ensures that the same query is run against the simulated ledger, ensuring that both will abort under the same conditions. This is the primary purpose for which $\mathcal{F}_{\mathsf{uSRS}}$ asks for a permutation on each invocation, despite only using it on the first, as well as why it supplies the identity of the calling party. □ □

As it is possible to construct (non-succinct) non-interactive zero-knowledge from a random oracle, we can remove the requirement on $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$ and instead rely on a random oracle $\mathcal{F}_{\mathsf{RO}}$ (formally described in subsection 3.11.3). As almost all constructions of Nakamoto-style ledgers are in the random oracle model, our usage of a low-level NIZK is not a major additional assumption.

Corollary 1. *For any updateable reference string scheme $\mathcal{S}$, it is possible to realise the pair of functionalities $(\mathcal{F}_{\mathsf{nakLedger}}^{\mathsf{ideal}}, \mathcal{W}_{\mathsf{delay}}(\delta, \mathcal{F}_{\mathsf{uSRS}}))$ in the $(\mathcal{F}_{\mathsf{nakLedger}}^{\mathsf{real}}, \mathcal{F}_{\mathsf{RO}})$-hybrid world, and in the presence of $\mathcal{G}_{\mathsf{clock}}$.*

Chapter 4

Mir-BFT: High-Throughput Robust BFT for Decentralized Networks

In this Chapter, we describe Mir-BFT, a basis for the flexible consensus toolkit in Hyperledger Fabric. Since April 2021, Mir-BFT is a Hyperledger Lab[1], developed open-source as a standalone Byzantine fault-tolerant consensus library, and is foreseen to be integrated as an ordering service in Hyperledger Fabric.

Mir-BFT is a robust Byzantine fault-tolerant (BFT) total order broadcast protocol aimed at maximizing throughput on wide-area networks (WANs), targeting deployments in decentralized networks, such as permissioned and Proof-of-Stake permissionless blockchain systems. Mir-BFT is developed open-source and is planned to be integrated in a major permissioned blockchain project.

Mir-BFT is the first BFT protocol that allows multiple leaders to propose request batches independently (i.e., parallel leaders), in a way that precludes request duplication attacks by malicious (Byzantine) clients, by rotating the assignment of a partitioned request hash space to leaders. As this mechanism removes a single-leader bandwidth bottleneck and exposes a computation bottleneck related to authenticating clients even on a WAN, our protocol further boosts throughput using a client signature verification sharding optimization. Our evaluation shows that Mir-BFT outperforms state-of-the-art and orders more than 60000 signed Bitcoin-sized (500-byte) transactions per second on a widely distributed 100 nodes, 1 Gbps WAN setup, with typical latencies of a few seconds. We also evaluate Mir-BFT under different crash and Byzantine faults, demonstrating its robustness.

Mir-BFT relies on classical BFT protocol constructs, which simplifies reasoning about its correctness. Specifically, Mir-BFT is a generalization of the celebrated and scrutinized PBFT protocol. In a nutshell, Mir-BFT follows PBFT "safety-wise", with changes needed to accommodate novel features restricted to PBFT liveness.

Mir-BFT PoC code, as described in this chapter, is available at https://github.com/hyperledger-labs/mirbft/tree/research.

4.1 Introduction

Background. Byzantine fault-tolerant (BFT) protocols, which tolerate malicious (Byzantine [LSP82]) behavior of a subset of nodes, have evolved from being a niche technology for tolerating bugs and intrusions to be the key technology to ensure consistency of widely deployed decentralized networks in which multiple mutually untrusted parties administer different nodes (such as in blockchain systems) [Vuk15, CDE+16, GHM+17]. Specifically, BFT protocols are considered to be an alternative (or complementing) to energy-intensive and slow Proof-of-Work (PoW) consensus protocols used in

[1] https://github.com/hyperledger-labs/mirbft

early blockchains including Bitcoin [Vuk15, GKW+16]. BFT protocols relevant to decentralized networks are consensus and total order (TO) broadcast protocols [CGR11], which establish the basis for state-machine replication (SMR) [Sch90a] and smart-contract execution [Woo16].

BFT protocols are known to be efficient on small scale (few nodes) in clusters (e.g., [AGK+15, KAD+07]), or to exhibit modest performance on large scale (thousands or more nodes) across WAN (e.g., [GHM+17]). Recently, considerable research effort (e.g., [Buc16,CNG18,GAG+19,YMR+19,MXC+16]) focused on maximizing BFT performance in medium-sized WAN networks (e.g., at the order of 100 nodes) as this deployment setting is highly relevant to different types of decentralized networks.

On the one hand, *permissioned* blockchains, such as Hyperledger Fabric [ABB+18a], are rarely deployed at scales above 100 nodes, yet use cases gathering dozens of organizations, which do not necessarily trust each other, are very prominent [Cor]. On the other hand, the setting of up to 100 nodes is also highly relevant in the context of large scale *permissionless* blockchains, in which anyone can participate, that use weighted voting (based e.g., on Proof-of-Stake (PoS) [BG17, KRDO17b] or delegated PoS (DPoS) [Ten]), or committee-voting [GHM+17], to limit the number of nodes involved in the critical path of the consensus protocol. With such weighted voting, the number of (relevant) nodes for PoS/DPoS consensus is typically in the order of a hundred [Ten], or sometimes even less [EOS19]. Related open-membership blockchain systems, such as Stellar, also run consensus among less than 100 nodes [LLM+19].

Challenges. Most of the BFT scalability research (e.g., [Buc16,CNG18,GAG+19,YMR+19]) aims at addressing the scalability issues that arise in classical leader-based BFT protocols, such as the seminal PBFT protocol [CL02]. In short, in a leader-based protocol, a leader, who is tasked with assembling a batch of requests (block of transactions) and communicating it to all other nodes, has at least $O(n)$ work, where n is the total number of nodes. Hence the leader quickly becomes a bottleneck as n grows.

A promising approach to addressing scalability issues in BFT is to allow multiple nodes to act as *parallel leaders* and to propose batches independently and concurrently, either in a coordinated, deterministic fashion [CNG18, MBS13], or using randomized protocols [DRZ18, MXC+16, L. 16]. With parallel leaders, the CPU and bandwidth load related to proposing batches are distributed more evenly. However, the issue with this approach is that parallel leaders are prone to wasting resources by proposing the same duplicate requests. As depicted in Table 4.1, none of the current BFT protocols that allow for parallel leaders deal with request duplication, which is straightforward to satisfy in single leader protocols. The tension between preventing request duplication and using parallel leaders stems from two important attacks that an adversary can mount and an efficient BFT protocol needs to prevent: (i) the *request censoring attack* by Byzantine leader(s), in which a malicious leader simply drops or delays a client's request (transaction), and (ii) the *request duplication attack*, in which Byzantine clients submit the exact same request multiple times.

To counteract request censoring attacks, a BFT protocol needs to allow at least $f + 1$ different leaders to propose a request (where f, which is typically $O(n)$, is the threshold on the number of Byzantine nodes in the system). Single-leader protocols (e.g., [CL02, YMR+19]), which typically rotate the leadership role across all nodes, address censoring attacks relatively easily. On changing the leader, a new leader only needs to make sure they do not repeat requests previously proposed by previous leaders.

With parallel leaders, the picture changes substantially. If a (malicious or correct) client submits the same request to multiple parallel leaders concurrently, parallel leaders will duplicate that request. While these duplicates can simply be filtered out after order (or after the reception of a duplicate, during ordering), the damage has already been done — excessive resources, bandwidth and CPU have been consumed. To complicate the picture, naïve solutions in which: (i) clients are requested to sequentially send to one leader at the time, or (ii) simply to pay transaction fees for each duplicate, do not help. In the first case, Byzantine clients mounting a request duplication attack are not required to respect

	Parallel Leaders	**Prevents Req. Duplication**
PBFT [CL02]	no	**yes**
BFT-SMaRt [BSA14]	no	**yes**
Aardvark [CWA+09]	no	**yes**
RBFT [AMQ13]	no	**yes**
Spinning [VCBL09]	no	**yes**
Prime [AAC+08]	no	**yes**
700 [AGK+15]	no	**yes**
Zyzzyva [KAD+07]	no	**yes**
SBFT [GAG+19]	no	**yes**
HotStuff [YMR+19]	no	(yes)
Tendermint [Buc16]	no	**yes**
BFT-Mencius [MBS13]	**yes**	no
RedBelly [CNG18]	**yes**	no
Hashgraph [L. 16]	**yes**	no
Honeybadger [MXC+16]	**yes**	no
BEAT [DRZ18]	**yes**	no
Mir (this work)	**yes**	**yes**

Table 4.1: Comparison of Mir to related BFT protocols. By *(yes)* we denote a property which can be satisfied with minor modifications to the protocol.

sending a request sequentially and, what is more, such a behavior cannot be distinguished from a correct client who simply sends a transaction multiple times due to asynchrony or network issues. Even though some blockchain systems, such as Hedera Hashgraph, charge transaction fees for every duplicate [Bai20], this approach penalizes correct clients when they resubmit a transaction to counteract possible censoring attacks, or a slow network. In more established decentralized systems, such as Bitcoin and Ethereum, it is standard to charge for the same transaction only once, even if it is submitted by a client more than once. In summary, with up to $O(n)$ parallel leaders, request duplication attacks may induce an $O(n)$-fold duplication of every single request and bring the effective throughput of *unique* requests to be inversely proportional to number of nodes n, practically voiding the benefits of using multiple leaders.

Contributions. This chapter presents Mir-BFT (or, simply, Mir[2]), a novel BFT total order broadcast (TOB) protocol that is the first to combine parallel leaders with robustness to attacks [CWA+09]. Mir is developed open-source as a part of Hyperledger Labs.

In particular, Mir precludes request duplication performance attacks, and addresses other notable performance attacks [CWA+09], such as the Byzantine leader straggler attack. Mir is further robust to arbitrarily long (yet finite) periods of asynchrony and is optimally resilient (requiring optimal $n \geq 3f+1$ nodes to tolerate f Byzantine faulty ones). On the performance side, Mir achieves the best throughput to date on public WAN networks, as confirmed by our measurements on up to 100 nodes. The following summarizes the main features of Mir, as well as contributions of this chapter:

- Mir allows multiple parallel leaders to propose batches of requests concurrently, in a sense multiplexing several PBFT instances into a single total order, in a robust way. As its main novelty, Mir partitions the request hash space across replicas, preventing request duplication, while rotating this partitioned assignment across protocol configurations (epochs), addressing the request censoring attack. Mir further uses *client signature verification sharding* throughput optimization to offload CPU, which is exposed as a bottleneck in Mir once we remove the single-leader bandwidth bottleneck using parallel leaders.
- Mir avoids "design-from-scratch", which is known to be error-prone for BFT [AGK+15, AGM+17]. Mir is a generalization of the well-scrutinized PBFT protocol [3], which Mir closely follows "safety-wise" while introducing important generalizations only affecting PBFT liveness (e.g., (multiple) leader election). This simplifies the reasoning about Mir correctness.
- We implement Mir in Go and run it with up to 100 nodes in a multi-datacenter WAN, as well as

[2]In a number of Slavic languages, the word *mir* refers to universally good, global concepts, such as *peace* and/or *world*.
[3]Mir variants based on other BFT protocols can be derived as well.

in clusters and under different faults, comparing it to state of the art BFT protocols. Our results show that Mir convincingly outperforms state of the art, ordering more than 60000 signed Bitcoin-sized (500-byte) requests per second (req/s) on a scale of 100 nodes on a WAN, with typical latencies of few seconds. In this setup, Mir achieves 3x the throughput of the optimistic Chain of [AGK+15], and more than an order of magnitude higher throughput than other state of the art single-leader BFT protocols. To put this into perspective, Mir's 60000+ req/s on 100 nodes on WAN is 2.5x the alleged peak capacity of VISA (24k req/s [Cap18]) and more than 30x faster than the actual average VISA transaction rate (about 2k req/s [Vuk15]).

Roadmap. The rest of the chapter is organized as follows. In Section 4.2, we define the system model and in Section 4.3 we briefly present PBFT (for completeness). In Section 4.4, we give an overview of Mir and changes it introduces to PBFT. We then explain Mir implementation details in Section 4.5. We further list the Mir pseudocode in Section 4.6. This is followed by Mir correctness proof in Section 4.7. Section 4.8 introduces an optimization tailored to large requests, such as the ones featured by Hyperledger Fabric. Section 4.9 gives evaluation details. Finally, Section 4.10 discusses related work and Section 4.11 concludes.

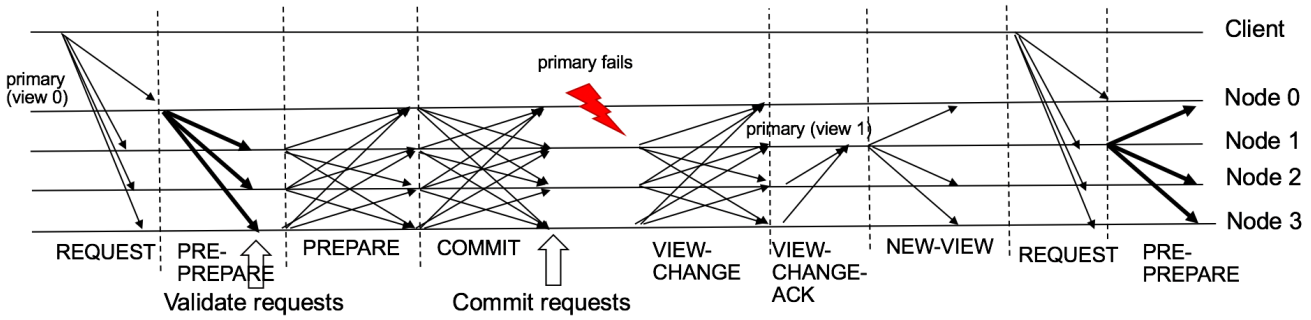


Figure 4.1: PBFT communication pattern and messages. Bottleneck messages are shown in **bold**.

4.2 System Model

We assume an eventually synchronous system [DLS88] in which the communication among *correct* processes can be fully asynchronous before some time denoted by GST, unknown to nodes, after which it is assumed to be synchronous. Processes are split into a set of n *nodes* (the set of all nodes is denoted by $Nodes$) and a set of *clients*. We assume a public key infrastructure in which processes are identified by their public keys; we further assume node identities are lexicographically ordered and mapped by a bijection to the set $[0 \dots n-1]$ which we use to reason about node identities. In every execution, at most f nodes can be *Byzantine* faulty (i.e., crash or deviate from the protocol in an arbitrary way), such that $n \geq 3f + 1$. Any number of clients can be Byzantine.

We assume an adversary that can control Byzantine faulty nodes but cannot break the cryptographic primitives we use, such as PKI and cryptographic hashes (we use SHA-256). $H(data)$ denotes a cryptographic hash of $data$, while $data_{\sigma_p}$ denotes $data$ signed by process p (client or node). Processes communicate through authenticated point-to-point channels (our implementation uses gRPC [gRP] over TLS, preventing man-in-the-middle and related attacks).

Nodes implement a BFT total order (atomic) broadcast service to clients. To broadcast request r, a client invokes $\text{BCAST}(r)$, with nodes eventually outputting $\text{COMMIT}(sn, r)$, such that the following properties hold:

- **P1 (Validity)** If a correct node commits r then some client broadcasted r.
- **P2 (Agreement/Total Order):** If two correct nodes commit requests r and r' with sequence number sn, then $r = r'$.
- **P3 (No duplication)** If a correct node commits request r with sequence numbers sn and sn', then $sn = sn'$.
- **P4 (Totality)** If a correct node commits request r, then every correct node eventually commits r.
- **P5 (Liveness)** If a correct client broadcasts request r, then some correct node p eventually commits r.

Note that P3 (No duplication) is a standard TOB property [CGR11] that most protocols can easily satisfy by filtering out duplicates *after* agreeing on request order, which is bandwidth wasting. Mir enforces P3 *without ordering duplicates*, using a novel approach to eliminate duplicates during agreement to improve performance and scalability.

4.3 PBFT and its Bottlenecks

Protocol	**PBFT** [CL02]	**Mir**
Client request authentication	vector of MACs (1 for each node)	signatures
Batching	no (or, 1 request per "batch")	yes
Multiple-batches in parallel	yes (watermarks)	yes (watermarks)
Round structure/naming	views	epochs
Round-change responsibility	view primary (round-robin across all nodes)	epoch primary (round-robin across all nodes)
No. of per-round leaders	1 (view primary)	many (from 1 to n epoch leaders)
No. of batches per round	unbounded	bounded (*ephemeral* epochs); unbounded (*stable* epochs)
Round leader selection	primary is the only leader	primary decides on epoch leaders (subject to constraints)
Request duplication prevention	enforced by the primary	hash space partitioning across epoch leaders (rotating)

Table 4.2: High level overview of the original PBFT [CL02] vs. Mir protocol structure.

We depict the PBFT communication pattern in Figure 4.1. PBFT proceeds in rounds called *views* which are led by the *primary*. The primary sequences and *proposes* a client's request (or a batch thereof) in a PRE-PREPARE message — on WANs this step is typically a network bottleneck. Upon reception of the PRE-PREPARE, other nodes validate the request, which involves, at least, verifying its authenticity (we say nodes *pre-prepare* the request). This is followed by two rounds of all-to-all communication (PREPARE and COMMIT messages), which are not bottlenecks as they leverage n links in parallel and contain metadata (request/batch hash) only. A node *prepares* a request and sends a COMMIT message if it gets a PREPARE message from a quorum ($n - f \geq 2f + 1$ nodes) that matches a PRE-PREPARE. Finally, nodes *commit* the request in total order, if they get a quorum of matching COMMIT messages.

The primary is changed only if it is faulty or if asynchrony breaks the availability of a quorum. In this case, nodes timeout and initiate a *view change*. View change involves communication among nodes in which they exchange information about the latest *pre-prepared* and *prepared* requests, such that the new primary, which is selected in round-robin fashion, must re-propose potentially committed requests under the same sequence numbers within a NEW-VIEW message (see [CL02] for details). The view-change pattern can be simplified using signatures [CL99b].

After the primary is changed, the system enters the new view and common-case operation resumes. PBFT complements this main common-case/view-change protocols with *checkpointing* (log and state compaction) and state transfer subprotocols [CL02].

4.4 Mir Overview

Mir is based on PBFT [CL02] (Sec. 4.3) — major differences are summarized in Table 4.2. In this section we elaborate on these differences, giving a high-level overview of Mir.

Request Authentication. While PBFT authenticates clients' requests with a vector of MACs, Mir uses signatures for request authentication to avoid concerns associated with "faulty client" attacks related to the MAC authenticators PBFT uses [CWA+09] and to prevent any number of colluding nodes, beyond f, from impersonating a client. However, this change may induce a throughput bottleneck, as per-request verification of clients' signatures requires more CPU than that of MACs. We address this issue by a signature verification sharding optimization described in Sec. 4.5.6.

Batching and Watermarks. Mir processes requests in *batches* (ordered lists of requests formed by a leader), a standard throughput improvement of PBFT (see e.g., [KAD+07, AGK+15]). Mir also retains request/batch *watermarks* used by PBFT to boost throughput. In PBFT, request watermarks, low and high, represent the range of request sequence numbers which the primary/leader can propose concurrently. While many successor BFT protocols eliminated watermarks in favor of batching (e.g, [KAD+07, BSA14, AGK+15]), Mir reuses watermarks to facilitate concurrent proposals of batches by *multiple parallel leaders*.

Protocol Round Structure. Unlike PBFT, Mir distinguishes between leaders and a primary node. Mir proceeds in *epochs* which correspond to *views* in PBFT, each epoch having a single *epoch primary* — a node deterministically defined by the epoch number, by round-robin rotation across all the participating nodes of the protocol.

Each epoch e has a set of *epoch leaders* (denoted by $EL(e)$), which we define as nodes that can sequence and propose batches in e (in contrast, in PBFT, only the primary is a leader). Within an epoch, Mir deterministically partitions sequence numbers across epoch leaders, such that all leaders can propose their batches simultaneously without conflicts. Epoch e transitions to epoch $e+1$ if (1) one of the leaders is suspected of failing, triggering a timeout at sufficiently many nodes (*ungracious epoch change*), or (2) a predefined number of batches $maxLen(e)$ has been committed (*gracious epoch change*). While the ungracious epoch change corresponds exactly to PBFT's view change, the gracious epoch change is a much more lightweight protocol.

Selecting Epoch Leaders. For each epoch, it is the *primary* who *selects the leaders* and reliably broadcasts its selection to all nodes. In principle, the primary can pick an arbitrary leader set as long as the primary itself is included in it. We evaluated a simple "grow on gracious, reduce on ungracious epoch" policy for leader set size. If primary i starts epoch e with a gracious epoch change it adds itself to the leader set of the preceding epoch $e-1$. If i starts epoch e with an ungracious epoch change and e' is the last epoch for which i knows the epoch configuration, i adds itself to the leader set of epoch e' and removes one node (not itself) for each epoch between e and e' (leaving at least itself in the leader set).

Moreover, in an epoch e where all nodes are leaders ($EL(e) = Nodes$), we define $maxLen(e) = \infty$ (i.e., e only ends if a leader is suspected). Otherwise, $maxlen(e)$ is a constant, pre-configured system parameter. We call the former *stable* epochs and the latter *ephemeral*.

More elaborate strategies for choosing epoch lengths and leader sets, which are outside the scope of this chapter, can take into account execution history, fault patterns, weighted voting, distributed randomness, or blockchain stake. Note that with a policy that constrains the leader set to only the epoch primary and makes every epoch stable, Mir reduces to PBFT.

Request Duplication and Request Censoring Attacks. Moving from single-leader PBFT to multi-leader Mir poses the challenge of request duplication. A simplistic approach to multiple leaders would be to allow any leader to add any request into a batch ([MBS13, CNG18, L. 16]), either in the common case, or in the case of client request retransmission. Such a simplistic approach, combined with a client sending a request to exactly one node, allows good throughput with no duplication only in the best case, i.e., with no Byzantine clients/leaders and with no asynchrony.

However, this approach does not perform well outside the best case, in particular with clients sending identical request to multiple nodes. A client may do so simply because it is Byzantine and performs the *request duplication* attack. However, even a correct client needs to send its request to at least $f + 1$ nodes (i.e., to $\Theta(n)$ nodes, when $n = 3f + 1$) in the worst case in *any* BFT protocol, in order to avoid Byzantine nodes (leaders) selectively ignoring the request (*request censoring* attack). Therefore, a simplistic approach to parallel request processing with multiple leaders [MBS13, CNG18, L. 16] faces attacks that can reduce throughput by factor of $\Theta(n)$, nullifying the effects of using multiple leaders.

Note the subtle but important difference between a duplication attack (submitting the same request to multiple replicas) and a DoS attack (submitting many different requests) that a Byzantine client can mount. A system can prevent the latter (DoS) by imposing per-client limits on incoming *unique* request rate. Mir enforces such a limit through client request watermarks. A duplication attack, however, is resistant to such mechanisms, as a Byzantine client is indistinguishable from a correct client with a less reliable network connection. We demonstrate the effects of these attacks in Section 4.9.5.

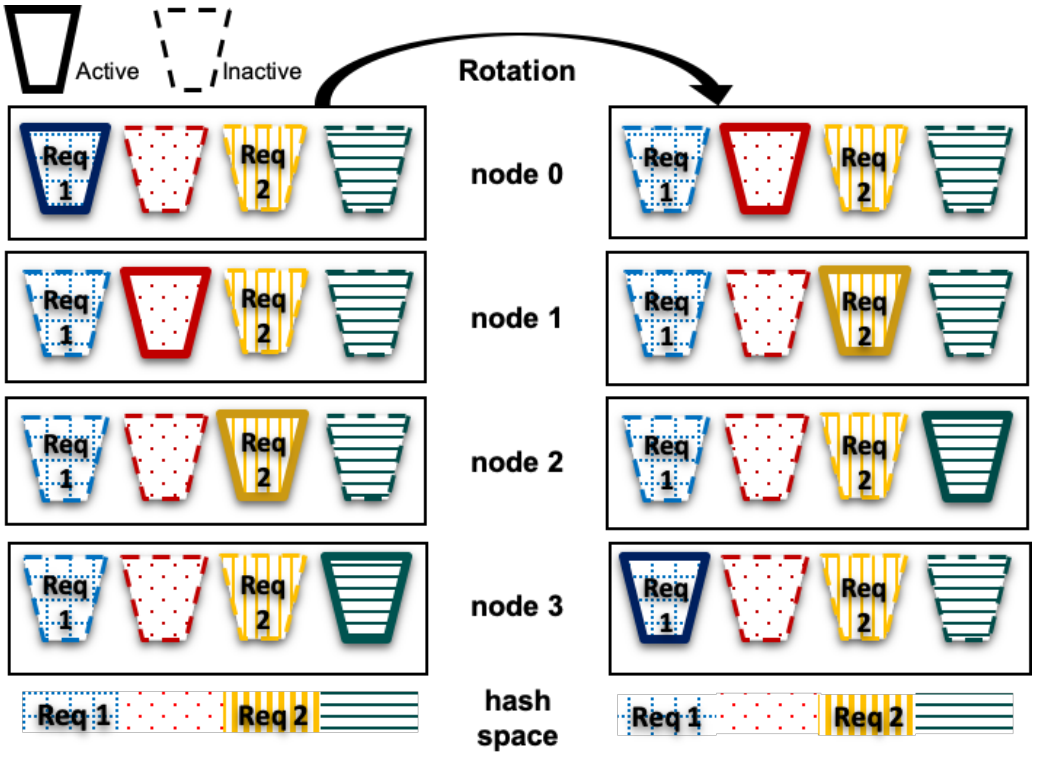


Figure 4.2: Request mapping in a stable epoch with $n = 4$ (all nodes are leaders): Solid lines represent the active buckets. Req. 1 is mapped to the first bucket, first active in node 1. Req. 2 is mapped to the third bucket, first active in node 3. Rotation redistributes bucket assignment across leaders.

Buckets and Request Partitioning. To cope with these attacks, Mir partitions the request hash space into buckets of equal size (number of buckets is a system parameter) and assigns each bucket to exactly one leader, allowing a leader to only propose requests from its assigned (*active*) buckets (preventing request duplication). For load balancing, Mir distributes buckets evenly (within the limits of integer arithmetics) to all leaders in each epoch. To prevent request censoring, Mir makes sure that every bucket will be assigned to a correct leader infinitely often. We achieve this by periodically redistributing the bucket assignment. Bucket re-distribution happens (1) at each epoch change (see Sec. 4.5.2) and (2) after a predefined number of batches have been ordered in a stable epoch (since a stable epoch might never end), as illustrated in Figure 4.2. Note that all nodes, while proposing only requests from their active buckets, still receive and store all requests (this can be optimized, see 4.5.1).

Parallelism. The Mir implementation (detailed in Sec. 4.5.10) is highly parallelized, with every *worker* thread responsible for one batch. In addition, Mir uses multiple gRPC connections among each pair of nodes which proves to be critical in boosting throughput in a WAN especially with a small number of nodes.

Generalization of PBFT and Emulation of Other BFT Protocols. Mir reduces to PBFT by setting $StableLeaders = 1$. This makes every epoch stable, hides bucket rotation (primary is the single leader) and makes every epoch change ungracious. Mir can also approximate protocols such as Tendermint [Buc16] and Spinning [VCBL09] by setting $StableLeaders > 1$, and fixing the maximum number of batches and leaders in every epoch to 1, making every epoch an ephemeral epoch and rotating leader/primary with every batch.

4.5 Mir Implementation Details

4.5.1 The Client

Upon BCAST(r), broadcasting request r, client c creates a message $\langle \text{REQUEST}, r, t, c \rangle_{\sigma_c}$. The message includes the client's timestamp t, a monotonically increasing sequence number that must be in a sliding window between the low and high *client watermark* $t_{c_L} < t \leq t_{c_H}$. Client watermarks in Mir allow multiple requests originating from the same client to be "in-flight", to enable high throughput without excessive number of clients. These watermarks are periodically advanced with the checkpoint mechanism described in Section 4.5.5, in a way which leaves no unused timestamps.

In principle, the client sends the REQUEST to all nodes (and periodically re-sends it to those nodes who have not received it, until request commits at at least $f + 1$ nodes). In practice, a client may start by sending its request to fewer than n nodes ($f + 1$ in our implementation) and only send it to the remaining nodes if the request has not been committed after a timeout.

4.5.2 Sequence Numbers and Buckets

Sequence Numbers. In each epoch e, a leader may only use a subset of e's sequence numbers for proposing batches. Mir partitions e's sequence numbers to leaders in $EL(e)$ in a round-robin way, using modulo arithmetic, starting at the epoch primary (see Fig. 4.3). We say that a leader *leads* sequence number sn when the leader is assigned sn and is thus expected to send a PRE-PREPARE for the batch with sequence number sn. Batches are proposed in parallel by all epoch leaders and are processed like in PBFT. Recall (from Table 4.2) that batch watermarking (not to be confused with client request watermarking from Sec. 4.5.1) allows the PBFT primary to propose multiple batches in parallel; in Mir, we simply extend this to multiple leaders.

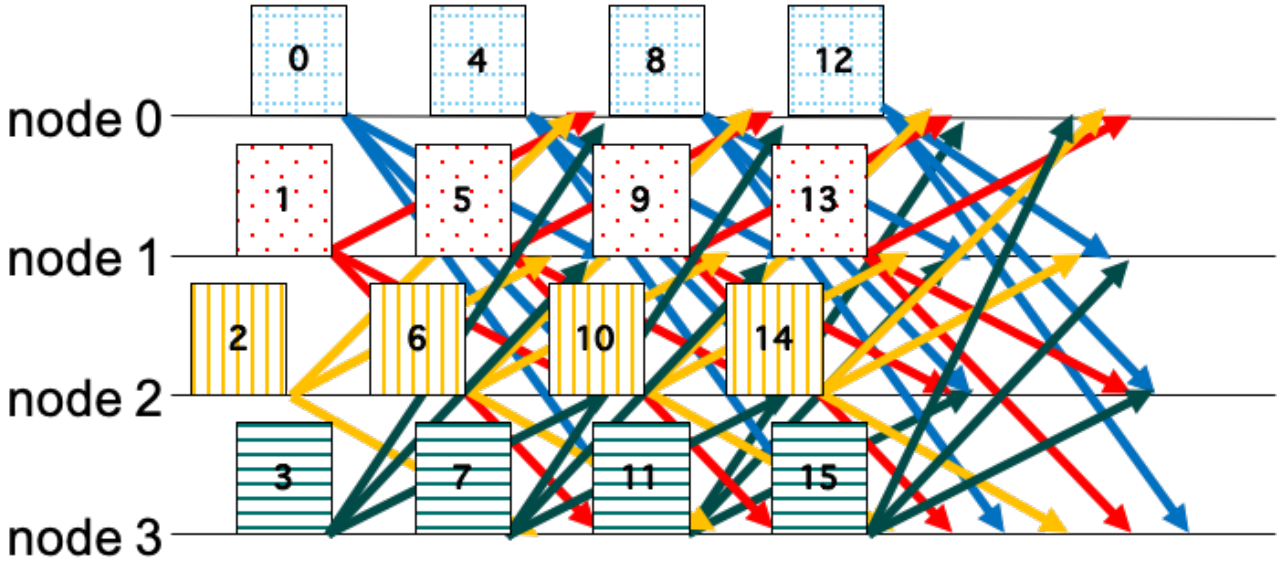


Figure 4.3: PRE-PREPARE messages in an epoch where all 4 nodes are leaders balancing the proposal load. Mir partitions batch sequence numbers among epoch leaders.

Buckets. In epoch $e = 0$, we assign buckets to leaders sequentially, starting from the buckets with the lowest hash values which we assign to epoch primary 0. For $e > 0$, the primary picks a set of consecutive buckets for itself (primary's *preferred buckets*), starting from the bucket which contains the *oldest* request it received; this is key to ensuring Liveness (P5, Sec. 4.2). Mir distributes the remaining buckets evenly and deterministically among the other leaders — this distribution is determined from an *epoch configuration* which the epoch primary reliably broadcasts and which contains preferred buckets and leader set selection (see Sec. 4.5.4). Buckets assigned to a leader are called its *active* buckets.

Additionally, if e is stable (when $maxLen(e) = \infty$ and thus no further epoch changes are guaranteed), leaders periodically (each time a pre-configured number of batches are committed) rotate the bucket assignment: leader i is assigned buckets previously assigned to leader $i + 1$ (in modulo n arithmetic). To prevent accidental request duplication (which could result in leader i being suspected and removed from the leader set), leader i waits to commit all "in-flight" batches before starting to propose its own batches. Other nodes do the same before pre-preparing batches in i's new buckets. In the example shown in Fig. 4.2, after the rotation, node 0 waits to commit all batches (still proposed by node 1) from its newly active red (second) bucket, before node 0 starts proposing new batches from the red (second) bucket.

4.5.3 Common Case Operation

REQUEST. In the common case, the protocol proceeds as follows. Upon receiving $\langle \text{REQUEST}, r, t, c \rangle_{\sigma_c}$ from a client, an epoch leader first verifies that the request timestamp is within the client's current watermarks $t_{C_L} < t \leq t_{C_H}$ and maps the request to the respective bucket by hashing the client timestamp and identifier $h_r = H(t||c)$. Each bucket is implemented as a FIFO queue. We do *not* hash the request payload, as this would allow a malicious client to target a specific bucket by adapting the request payload, mounting load imbalance attacks. If the request falls into the leader's active bucket, the leader also verifies the client's signature on REQUEST. A node i discards r if i already preprepared a batch containing r or if r is already pending at i (we call a valid request pending if it has been received by i but not yet committed).

PRE-PREPARE. Once leader i gathers enough requests in its current active buckets, or if timer T_{batch} expires (since the last batch was proposed by i), i adds the requests from the current active buckets in a batch, assigns its next available sequence number sn to the batch (provided sn is within batch watermarks) and sends a PRE-PREPARE message. If T_{batch} time has elapsed and no requests are available, i sends a PRE-PREPARE message with an empty batch.. This guarantees progress of the protocol under low load.

A node j accepts a PRE-PREPARE (we say *preprepares* the batch and the requests it contains), with sequence number sn for epoch e from node i provided that: (1) the epoch number matches the local epoch number and j did not preprepare another batch with the same e and sn, (2) node i is in $EL(e)$, (3) node i leads sn, (4) the batch sequence number sn in the PRE-PREPARE is between a low watermark and high batch watermark: $w < sn \leq W$, (5) none of the requests in the batch have already been *preprepared*, (6-8) every request in the batch: (6) has timestamp within the current client's watermarks, (7) belongs to one of i's active buckets, and (8) has a signature which verifies against client's id (i.e., corresponding public key).

Conditions (1)-(4) are equivalent to checks done in PBFT, whereas conditions (5)-(8) differ from PBFT. Condition (5) is critical for enforcing No Duplication (Property P3, Sec. 4.2). Conditions (6) (allowing clients to send more than one request concurrently) and (7) (prohibiting malicious leaders to propose requests outside their buckets) are performance related. Condition (8) is the key to Validity (Property P1). As this step may become a CPU bottleneck if performed by all nodes, we use signature sharding as an optimization (see Sec. 4.5.6).

The Rest. If node j preprepares the batch, j sends a PREPARE and the protocol proceeds exactly as PBFT. Otherwise, j ignores the batch (which may eventually lead to j entering epoch change). Upon committing a batch, j removes all requests present in the committed batch from j's pending queues.

4.5.4 Epoch Change

Locally, at node j, epoch e can end *graciously*, by exhausting all $maxLen(e)$ sequence numbers, or *ungraciously*, if an epoch change timer (corresponding to the PBFT view change timer) at j expires. In the former (gracious) case, a node simply starts epoch $e+1$ (see also Sec. 4.5.4) when it: (1) locally commits all sequence numbers in e, and (2) reliably delivers epoch configuration for $e+1$. In the latter (ungracious) case, a node first enters an epoch change subprotocol (Sec. 4.5.4) for epoch $e+1$.

It can happen that some correct nodes finish e graciously and some others not. Such (temporary) inconsistency may prevent batches from being committed in $e+1$ even if the primary of $e+1$ is correct. However, such inconsistent epoch transitions are eventually resolved in subsequent epochs, analogously to PBFT, when some nodes complete the view change subprotocol and some do not (due to asynchrony). As we show in in Section 4.7.5, the liveness of Mir is not violated.

Epoch Change Subprotocol

The epoch change subprotocol is triggered by epoch timeouts due to asynchrony or failures and generalizes PBFT's view change subprotocol. Similarly to PBFT, after commiting batch sn in epoch e a node resets and starts an epoch-change timer $ecTimer$ expecting to commit batch $sn+1$.

If an $ecTimer$ expires at node i, i enters the epoch-change subprotocol to move from epoch e to epoch $e+1$.

In this case, i sends an EPOCH-CHANGE message to the primary of epoch $e+1$. An EPOCH-CHANGE message follows the structure of a PBFT VIEW-CHANGE message (page 411, [CL02]) with the difference that it is signed and that there are no VIEW-CHANGE-ACK messages exchanged (to streamline and simplify the implementation similarly to [CL99a]). The construction of a NEW-EPOCH message (by the primary of $e+1$) proceeds in the same way as the PBFT construction of a

NEW-VIEW message. A node starts epoch $e+1$ by processing the NEW-EPOCH message the same way a node starts a new view in PBFT by processing a NEW-VIEW message.

However, before entering epoch $e+1$, each correct node *resurrects* potentially pre-prepared but uncommitted requests from previous epochs that are not reflected in the NEW-EPOCH message. This is required to prevent losing requests due to an epoch change (due to condition (5) in pre-preparing a batch — Sec. 4.5.3), as not all batches that were created and potentially preprepared before the epoch change were necessarily delivered when starting the new epoch. Resurrecting a request involves each correct node: (1) returning these requests to their corresponding bucket queues, and (2) marking the requests as not preprepared. This allows proposing such requests again. Together with the fact that clients make sure that every correct replica eventually receives their request (Sec. 4.5.1), this helps guarantee Liveness (P5).

Starting a New Epoch

Every epoch e be it gracious or ungracious, starts by the primary reliably broadcasting (using Bracha's classic 3-phase algorithm [BT85]) the *epoch configuration* information[4] containing: (1) $EL(e)$, the set of epoch leaders for e, and (2) identifiers of primary's preferred buckets (that the primary picks for itself), which the primary selects based on the oldest request pending at the primary.

Before starting to execute participate in epoch e (including processing a potential NEW-EPOCH message for e) a node i first waits to reliably deliver the epoch e configuration. In case of gracious epoch change, node i also waits to locally committing all "in-flight" batches pertaining to $e-1$.

4.5.5 Checkpointing (Garbage Collection)

Exactly as in PBFT, Mir uses a checkpoint mechanism to prune the message logs. After a node i commits all batches with sequence numbers up to and including sn_C, where sn_C is divisible by predefined configuration parameter C, i sends a $\langle \text{CHECKPOINT}, sn_C, H(sn'_C)\rangle\sigma_i$ message to all nodes, where sn'_C is the last checkpoint and $H(sn'_C)$ is the hash of the batches with sequence numbers sn in range $sn'_C \leq sn < sn_C$. Each node collects checkpoint messages until it has $2f+1$ matching ones (including its own), constituting a *checkpoint certificate*, and persists the certificate. At this point, the checkpoint is *stable* and the node can discard the common-case messages from its log for sequence numbers lower than sn.

Mir advances batch watermarks at checkpoints like PBFT does. Clients' watermarks are also possibly advanced at checkpoints, as the state related to previously delivered requests is discarded. For each client c, the low watermark t_{c_L} advances to the highest timestamp t in a request submitted by c that has been delivered, such that all requests with timestamp $t' < t$ have also been delivered. The high watermark advances to $t_{c_H} = t_{c_L} + w_c$, where w_c is the length of the sliding window.

4.5.6 Signature Verification Sharding (SVS)

To offload CPU during failure-free execution (in stable epochs), we implement an optimization where not all nodes verify all client signatures. For each batch, we distinguish $f+1$ *verifier* nodes, defined as the $f+1$ lexicographic (modulo n) successors of the leader proposing the batch. Only the verifiers verify client signatures in the batch on reception of a PRE-PREPARE message (condition (8) in Sec. 4.5.3). Furthermore, we modify the Mir (and thus PBFT) common-case protocol such that a node does not send a COMMIT before having received a PREPARE message from *all* $f+1$ verifiers (in addition to f other nodes and itself). This maintains Validity, as at least one correct node must have verified client's signature. This way, however, if even a single verifier is faulty, SVS may prevent a batch

[4]We optimize the reliable broadcast of an epoch configuration using piggybacking on other protocol messages where applicable.

from being committed. Therefore, we only apply this optimization in stable epochs where all nodes are leaders. In case (ungracious) epoch change occurs, reducing the size of the leader set, Mir disables SVS.

4.5.7 State Transfer

Nodes can temporarily become unavailable, either due to asynchrony, or due to transient failures. Upon recovery/reconnection, nodes must obtain several pieces of information before being able to actively participate in the protocol again. Mir state transfer is similar to that of PBFT, and here we outline the key aspects of our implementation.

To transfer state, nodes need to obtain current epoch configuration information, the latest stable checkpoint (which occurred at sequence number h), as well as information concerning batches having sequence numbers between $h + 1$ and the latest sequence number. Nodes also exchange information about committed batches.

The state must, in particular, contain two pieces of information: (1) the current epoch configuration, which is necessary to determine the leaders from which the node should accept proposals, and (2) client timestamps at the latest checkpoint, which are necessary to prevent including client requests that have already been proposed in future batches.

A node i in epoch e goes to state transfer when i receives common-case messages from $f + 1$ other nodes with epoch numbers higher than e, and i does not transition to $e + 1$ for a certain time. Node i obtains this information by broadcasting a $\langle HELLO, ne_i, c_i, b_i \rangle$ message, where ne_i is the latest NEW-EPOCH message received by i, c_i is the node's last stable checkpoint, and b_i is the last batch i delivered. Upon receipt of a $HELLO$ message, another node j replies with its own $HELLO$ message, as well as with any missing state from the last stable checkpoint and up to its current round n.

From the latest stable checkpoint, a node can derive the set of $2f + 1$ nodes which signed this stable checkpoint. This also allows a node to transfer missing batches even from one out of these $2f + 1$ nodes, while receiving confirmations of hashes of these batches from f additional nodes (to prevent ingress of batches from a Byzantine node).

We perform further optimizations in order to reduce the amount of data that needs to be exchanged in case of a state transfer. First, upon reconnecting, nodes announce their presence but wait for the next stable checkpoint after state transfer before actively participating in the protocol again. This enables us to avoid transferring the entire state related to requests following the preceding stable checkpoint. Second, the amount of data related to client timestamps that needs to be transmitted can be reduced through only exchanging the root of the Merkle tree containing the client timestamps, with the precise timestamps being fetched on a per-need basis.

4.5.8 Membership Reconfiguration

While details of membership reconfiguration are outside of the scope of this chapter, we briefly describe how Mir deals with adding/removing clients and nodes. Such requests, called *configuration* requests are totally ordered like other requests, but are tagged to be interpretable/executed by nodes. As Mir processes requests out of order (just like PBFT), configuration requests cannot be executed right after committing a request as the timing of commitment might diverge across nodes resulting in non-determinism. Instead, configuration requests are taken into account only at checkpoints and more specifically all configuration requests ordered between checkpoints $k - 1$ and k, take effect only after checkpoint $k + 1$.

4.5.9 Durability (Persisting State)

By default, Mir implementation does not persist state or message logs to stable storage. Hence, a node that crashes might recover in a compromised state — however, such a node does not participate in the

protocol until the next stable checkpoint which effectively restores the correct state. While we opted for this approach assuming that for few dozens of nodes simultaneous faults of up to a third of them will be rare, for small number of nodes the probability of such faults grows and with some probability might exceed threshold f. Therefore, we optionally persist state pertaining to *sent* messages in Mir, which is sufficient for a node to recover to a correct state after a crash.

We also evaluated the impact of durability with 4 nodes, in a LAN setting, where it is mostly relevant due to small number of nodes and potentially collocated failures, using small transactions. We find that durability has no impact on total throughput, mainly due to the fact that persisted messages are amortized due to batching, Mir parallel architecture and the computation-intensive workload. However, average request latency increases by roughly 300ms.

4.5.10 Implementation Architecture

We implemented Mir in Go. Our implementation is multi-threaded and inspired by the *consensus-oriented parallelism* (COP) architecture previously applied to PBFT to maximize its throughput on multicore machines [BDK15]. Specifically, in our implementation, a separate thread is dedicated to managing each batch during the common case operation, which simplifies Mir code structure and helps maximize performance. We further parallelize computation-intensive tasks whenever possible (e.g., signature verifications, hash computations). The only communication in common case between Mir threads pertains to request duplication prevention (rule (6) in accepting PRE-PREPARE in Sec. 4.5.3) — the shared data structures for duplication prevention are hash tables, synchronized with per-bucket locks; instances that handle requests corresponding to different leaders do not access the same buckets. The only exception to the multi-threaded operation of Mir is during an ungracious epoch-change, where a designated thread (Mir Manager) is responsible for stopping worker common-case threads and taking the protocol from one epoch to the next. This manager thread is also responsible for sequential batch delivery and for checkpointing, which, however, does not block the common-case threads processing batches.

Our implementation also parallelizes network access using a configurable number of independent network connections between each pair of nodes. This proves to be critical in boosting Mir performance beyond seeming bandwidth limitations in a WAN that stem from using a single TCP/TLS connection.

In addition to multiple inter-node connections, we use an independent connection for handling client requests. As a result, the receipt of requests is independent of the rest of the protocol — we can safely continue to receive client requests even if the protocol is undergoing an epoch change. Our implementation can hence seamlessly use, where possible, separate NICs for client's requests and inter-node communication to address DoS attacks [CWA+09].

4.6 Pseudocode

In this section we introduce Mir pseudocode. We first present PBFT [CL02] pseudocode to demonstrate the common message flow in the common case of the two protocols.

Each node executes its own instance of the algorithm described by the pseudocode. The node atomically executes each **upon** block exactly once for each assignment of values satisfying the block's triggering condition.

For better readability we do not include batching in the pseudocode. Implementing batching is trivial by replacing requests with batches of requests, except request handling (lines 53-60). Moreover, whenever appropriate, instead of performing a request-specific action on a batch, we perform this action on all requests in a batch, like request validity checks in PRE-PREPARE (lines 74-77) and request resurrection (lines 138-146). In the context of request-specific validity checks, we consider the whole batch invalid if any of the contained requests fails its validity check. Finally, condition on line 62 should be replaced with checking if there exist enough requests for a batch.

Moreover, for readability, the pseudocode does not include a batch timeout which ensures that even with low load leaders continuously send batches (even if empty) to drive the checkpoint protocol and so that EpochChangeTimeout does not fire.

Algorithm 1 Common

```
1: function IsPrimary(i, v) :
2:     return i = v  mod N;
3:
4: function Valid(v, n) :
5:     if (lv = v)  and (w <= n < W) then
6:         return True;
7:     else
8:         return False;
9:
10: function GetOldest(S) :
11:     Returns the oldest entry in set S \ Preprepared.
12:
```

Algorithm 2 PBFT [CL02]

```
1: import Common
2: import PbftViewChange
3:
4: Parameters:
5:     id
6:     // The node identity
7:     f
8:     // Number of faults tolerated
9:     RequestTimeout
10:    // Timeout to prevent waiting indefinitely for q request to commit
11:    w
12:    // Low watermark, advances at checkpoints
13:    W
14:    // High watermark, advances at checkpoints
15:
16: Struct Request contains
17:    bytes o
18:    // Request payload
19:    int t
20:    // Client timestamp
21:    bytes c
22:    // Client public key (ID)
23:
24: Init:
25:    lv ← 0
26:    // Local view number
27:    next ← 0
28:    // The next available sequence number
29:    R ← ∅
30:    // The set of received requests
31:    Preprepare_msgs ← {}
32:    // A map from (view, sequence number) pairs to PRE-PREPARE messages, initially ⊥
33:    Prepare_msgs ← {}
34:    // A map from (view, sequence number) pairs to a set of unique PREPARE messages
35:    Commit_msgs ← {}
36:    // A map from (view, sequence number) pairs to a set of unique COMMIT messages
37:    RequestTimeouts ← {}
38:    // A map from requests to timers
39:
```

Algorithm 2 PBFT (continues)

upon receiving $r \leftarrow \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
    such that $SigVer(r, \sigma_c, c)$
    and not (r' **in** R s.t. $r'.c = r.c$ **and** $r'.t \neq r.t$) **do**
    $R \leftarrow r \cup R$
    $RequestTimeouts[r] \leftarrow$ **schedule** $RequestTimeout$

upon receiving $|R| > 0$ **and** $w <= next < W$
 and common.IsPrimary(id, lv) **do**
$r \leftarrow common.GetOldest(R)$
Send $\langle \text{PRE-PREPARE}, lv, next, r, id \rangle$ to all nodes
$next \leftarrow next + 1$
upon receiving $pp \leftarrow \langle \text{PRE-PREPARE}, v, n, r, i \rangle$
    such that $common.Valid(v, n)$
    and $common.IsPrimary(i, v)$
    and $Preprepare\_msgs[v, n] = \perp$
    and r **in** R **do**
    $Preprepare\_msgs[v, n] \leftarrow pp$
    send $\langle \text{PREPARE}, v, n, D(r), id \rangle$ to all nodes

upon receiving $p \leftarrow \langle \text{PREPARE}, v, n, D(r), i \rangle$
    such that $D(Preprepare\_msgs[v, n].r) = D(r)$
    and $common.Valid(n, v)$ **do**
    $Prepare\_msgs[v, n] \leftarrow Prepare\_msgs[v, n] \cup \{p\}$

upon $|Prepare\_msgs[lv, n]| = 2f + 1$ **do**
    $r \leftarrow Preprepare\_msgs[lv, n].r$
    send $\langle \text{COMMIT}, lv, n, D(r), id \rangle$ to all nodes

upon receiving $c \leftarrow \text{COMMIT}, v, n, D(r), i \rangle$
    such that $D(Preprepare\_msgs[v, n].r) = D(r)$
    and $common.Valid(n, v)$ **do**
    $Commit\_msgs[v, n] \leftarrow Commit\_msgs[v, n] \cup \{c\}$

upon $|Commit\_msgs[lv, n]| = 2f + 1$ **do**
    $r \leftarrow Preprepare\_msgs[v, n].r$
    $R \leftarrow R \setminus r$
    $Deliver(n, r)$
    cancel $RequestTimeouts[r]$

upon $RequestTimeout$ **do**
    $lv \leftarrow lv + 1$
    $PbftViewChange.ViewChange()$

Algorithm 3 Mir initialization

```
1: import Common
2: import PbftViewChange
3: import ReliableBroadcast
4:
5: Parameters:
6:     id
7:     // The node identity
8:     f
9:     // Number of faults tolerated
10:    EpochChangeTimeout
11:    // Timeout for epoch change
12:    w
13:    // Low watermark, advances at checkpoints
14:    W
15:    // High watermark, advances at checkpoints
16:    NumBuckets
17:    // Number of buckets
18:    BucketsPerLeader
19:    // The number of buckets per leader when all nodes are leaders
20:    RotationPeriod
21:    // Bucket rotation period
22:    EphemeralEpLen
23:    // Number of sequence numbers in an ephemeral epoch
24:
25: Struct Request contains
26:    o
27:    // Request payload
28:    t
29:    // Client timestamp
30:    c
31:    // Client struct
32:
33: Struct Client contains
34:    pk
35:    // Client public key (ID)
36:    H
37:    // Client high watermark, advances at checkpoint
38:    L
39:    // Client low watermark, advances at checkpoint
40:
41: Struct EpochConfig contains
42:    First
43:    // First sequence number of the epoch
44:    Last
45:    // Last sequence number of the epoch
46:    Leaders
47:    // List of leaders of the epoch
48:    PrimaryBuckets
49:    // Buckets the primary chose for itself
50:
51: Init:
52:    le ← 0
53:    // Local epoch number
54:    next ← id
55:    // The next available sequence number
56:    Buckets ← Set of NumBuckets empty buckets
57:
58:    // Each bucket is a FIFO queue of received requests
59:    Preprepare_msgs ← {}
60:    // A map from (epoch, sequence number) pairs to PRE-PREPARE messages
61:    Prepare_msgs ← {}
62:    // A map from (epoch, sequence number) pairs to a set of unique PREPARE messages
63:    Commit_msgs ← {}
```

Algorithm 4 PBFT ViewChange

```
1: import Common
2:
3: Parameters:
4:     N
5:     // Number of nodes
6:     f
7:     // Number of faults
8:     id
9:     // The node identity
10:    lv
11:    // Local view number
12:    P
13:    // Map form sequence number
14:
15:    // to Entry struct for the latest prepared request in previous views
16:    Q
17:    // Map form sequence number
18:
19:    // to all Entry structs for a unique pre-prepared request in previous views
20:    C
21:    // Local checkpoints
22:    h
23:    // Latest stable checkpoint
24:
25: Init:
26:    Sset ← {}
27:    // A map from node id to ViewChange message
28:    Xset ← {}
29:    // A map from sequence number to selected value
30:    cp ← ⊥
31:    // The highest stable checkpoint available at f+1 nodes
32:
33: Struct Request contains
34:    n
35:    // Sequence number
36:    d
37:    // Request digest
38:    v
39:    // View
40:
41: upon receiving  m ← VIEWCHANGE, v, h, C, P, Q, i, σ_i⟩
42:     such that SigVer(m, σ_i, i.pk)
43:    V[i] ← m
44:    if |Sset| ≥ 2f + 1
45:        CalculateHighCheckpoint(Sset)
46:        CalculateXset(Sset) [5]
47:        if Xset ≠ {}
48: // If the Xset is successfully calculated
49:            send ⟨NEWVIEW, v, Sset, Xset, cp, id, σ_id⟩ to all nodes
50:        end if
51:    end if
52:
53: upon receiving  m ← NEWVIEW, v, S, X, cp′, i, σ_i⟩
54:     such that SigVer(m, σ_i, i.pk)
55:    CalculateHighCheckpoint(S)
56:    CalculateXset(S)
57:    if Xset = X and cp = cp′
58: // Verify NEWVIEW
59:        for all (n, r) ∈ Xset do
60:            send ⟨PREPARE, v, n, D(r), id⟩ to all nodes
61:        end for
62:    end if
63:
```

Algorithm 4 Mir (continues)

82: **upon receiving** $r \leftarrow \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$
83: **such that** $SigVer(m, \sigma_c, c.pk)$
84: **and** $r.c.L <= r.t < r.c.H$
85: **and** $r \notin Preprepared$ **do**
86: $bucket \leftarrow GetBucket(H(t||c.pk))$
87: **if** $\nexists r' \in bucket : r'.c = r.c \wedge r'.t = r.t$
88: $bucket.append(r)$
89: **end if**
90:
91: **upon** $|ActiveBuckets(i, le, next)| > 0$
92: **and** $w <= n < W$
93: **and** $ActiveRotation(le, n)$
94: **and** $n \leq EpochConfig[le].Last$ **do**
95: $r \leftarrow common.GetOldest(ActiveBuckets(i, le, next))$
96: send $\langle \text{PRE-PREPARE}, le, n, r, id \rangle$ to all nodes
97: $next \leftarrow next + |EpochConfig[le].Leaders)|$
98:
99: **upon receiving** $pp \leftarrow \langle \text{PRE-PREPARE}, e, n, r, i \rangle$
100: **such that** $common.Valid(e, n)$
101: **and** $IsLeader(i, e, n)$
102: **and** $Preprepare\_msgs[e, n] = \perp$)
103: **and** $r.c.L <= r.t < r.c.H$
104: **and** $H(r.o||r.t||r.c.pk)$ **not in** $Preprepared$
105: **and** $H(r.t||r.c.pk)$ **in** $ActiveBuckets(i, e, n)$
106: **and** $SigVer((r, r.\sigma_c, c.pk)$ **do**
107: $Preprepared \leftarrow Preprepared \cup \{r\}$
108: $Preprepare\_msgs[e, n] \leftarrow pp$
109: send $\langle \text{PREPARE}, v, n, D(r), id \rangle$ to all nodes
110:
111: **upon receiving** $p \leftarrow \langle \text{PREPARE}, e, n, D(r), i \rangle$
112: **such that** $D(Preprepare\_msgs[e, n].r) = D(r)$
113: **and** $common.Valid(e, v)$ **do**
114: $Prepare\_msgs[e, n] \leftarrow Prepare\_msgs[e, n] \cup \{p\}$
115:
116: **upon** $|Prepare\_msgs[le, n]| = 2f + 1$ **do**
117: $r \leftarrow Preprepare\_msgs[e, n].r$
118: send $\langle \text{COMMIT}, le, n, D(r), id \rangle$ to all nodes
119:
120: **upon receiving** $c \leftarrow \langle \text{COMMIT}, e, n, D(r), i \rangle$
121: **such that** $D(Preprepare\_msgs[e, n].r) = D(r)$
122: **and** $common.Valid(e, v)$ **do**
123: $Commit\_msgs[e, n] \leftarrow Commit\_msgs[e, n] \cup \{c\}$
124:
125: **upon** $|Commit\_msgs[e, n]| = 2f + 1$ **do**
126: $r \leftarrow Preprepare\_msgs[e, n].r$
127: $committed[e, n] \leftarrow r$
128: $GetBucket(H(r.t||r.c.pk)).remove(r)$
129:
130: **upon** $committed[le, n] \neq \perp$ **and** $delivered[n - 1]$ **do**
131: $Deliver(n, r)$
132: $delivered[n] \leftarrow True$
133: **reset** EpochChangeTimeout
134:
135: **upon** $EpochChangeTimeout$ **do**
136: $PBFTViewChange.ViewChange()$
137: `// PBFT view change`
138:

139: **upon** delivered[EpochConfig[e].Last]
140: **and** common.IsPrimary(id, e+1) **do**
141: $EpochConfig[e + 1].Leaders \leftarrow EpochConfig[e].Leaders \cup \{id\}$
142: $EpochConfig[e + 1].PrimaryBuckets$
143: $\leftarrow \lceil NumBuckets/Nodes \rceil$ buckets containing the oldest requests
144: $EpochConfig[e + 1].First \leftarrow EpochConfig[e].Last + 1$

Algorithm 4 Mir (continues)

upon sending PBFT NEW-EPOCH message for epoch $e+1$ **do**
    $EpochConfig[e+1].Leaders \leftarrow ShrinkingLeaderset(e+1, id)$
    $EpochConfig[e+1].PrimaryBuckets$
        $\leftarrow \lceil NumBuckets/Nodes \rceil$ buckets containing the oldest requests
    $EpochConfig[e+1].First \leftarrow EpochConfig[e].Last + 1$
    $EpochConfig[e+1].Last \leftarrow EpochConfig[e+1].First + ephemeralEpLen$
    $ReliableBroadcast.Broadcast(EpochConfig[e+1], e+1)$

upon $ReliableBroadcast.Delivered(EpochConfig, e)$ **and** $le = e$ **do**
    $EpochConfig[e] \leftarrow EpochConfig$
    if $\exists k : EpochConfig[e].Leaders[k] = id$ **then**
        $next \leftarrow EpochConfig[e].First + k$
    $ActiveBucketAssignment(e, EpochConfig)$

upon sending or receiving PBFT NEW-EPOCH message **do**
    for all $r \in Preprepared$ **do**
        if r not in NEW-VIEW **then**
            $Preprepared \leftarrow Preprepared \setminus \{r\}$
    end for
    for all $r \in$ PBFT NEW-VIEW **do**
        $Preprepared \leftarrow Preprepared \cup \{r\}$
    end for

function $IsLeader(i, e, n)$ **:**
    // Returns $True$ if i is leader of n in epoch e
    if i **in** $EpochConfig[e].Leaders$ **then**
        return $(EpochConfig[e].First + n \equiv i \mod |EpochConfig[e].Leaders|$
    else
        return False

function $GetBucket(hash)$ **:**
    Returns the bucket containing requests r such that $H(r.t||r.c.pk) = hash$.

function $ActiveBucketAssignment(e, EpochConfig)$ **:**
    Evenly partition $Buckets \setminus EpochConfig.PrimaryBuckets$
    among $EpochConfig.Leaders \setminus \{i : IsPrimary(i, e)\}$

function $ActiveBuckets(i, e, n)$ **:**
    Returns the union of buckets which are active for node i in epoch e and sequence number n

// ActiveRotation returns true if all the sequence numbers from the previous rotation are delivered
function $ActiveRotation(e, n)$ **:**
    $period \leftarrow RotationPeriod$
    $rotation \leftarrow \lceil n - (EpochConfig[e-1].Last)/period \rceil$
    return $delivered[EpochConfig[e-1].Last + (rotation - 1) * period]$

function $ShrinkingLeaderset(e, i)$ **:**
    $e_{last} \leftarrow$ the last epoch for which i has the configuration
    $Leaders \leftarrow EpochConfig[e_{last}].Leaders \cup \{i\}$
    $RemovedLeaders \leftarrow$ a random set of $min((e' - e), 1)$ nodes from $EpochConfig[e_{last}].Leaders \setminus \{i\}$
    return $Leaders \setminus RemovedLeaders$

4.7 Mir Correctness

In this section we outline Mir correctness proof, proving TOB properties as defined in Section 4.2. We pay particular attention to Liveness (Section 4.7.5), as we believe it is the least obvious out of four Mir TOB properties to a reader knowledgeable in PBFT. Where relevant, we also consider the impact of the signature verification sharding (SVS) optimization (Sec. 4.5.6).

We define a function for assigning request sequence numbers to individual requests, output by COMMIT indication of TOB as follows. For batch with sequence number $sn \geq 0$ committed by a correct node i, let S_{sn} be the total number of requests in that batch (possibly 0). Let r be the k^{th} request that a correct node commits in a batch with sequence number $sn \geq 0$. Then, i outputs COMMIT$(k + \sum_{j=0}^{sn-1} S_j, r)$, i.e., node i commits r with sequence number $k + \sum_{j=0}^{sn-1} S_j$.

4.7.1 Validity (P1)

(P1) **Validity:** If a correct node commits r then some client broadcasted r.

Proof (no SVS). We first show Validity holds, without signature sharding. If a correct node commits r then at least $n - f$ correct nodes sent COMMIT for a batch which contains r, which includes at least $n - 2f \geq f + 1$ correct nodes (Sec. 4.3). Similarly, if a correct node sends COMMIT for a batch which contains r, then at least $n - 2f \geq f + 1$ correct nodes sent PREPARE after pre-preparing a batch which contains r (Sec. 4.5.3). This implies at least $f + 1$ correct nodes executed Condition (8) in Sec. 4.5.3 and verified client's signature on r as correct. Validity follows. □

Proof (with SVS). With signature verification sharding (Sec. 4.5.6), clients' signatures are verified by at least $f + 1$ *verifier* nodes belonging to the leader set, out of which at least one is correct. As no correct node sends COMMIT before receiving PREPARE from all $f + 1$ *verifier* nodes (Sec. 4.5.6), no request which was not correctly signed by a client can be committed — Validity follows. □

4.7.2 Agreement (Total Order) (P2)

(P2) **Agreement:** If two correct nodes commit requests r and r' with sequence number sn, then $r = r'$.

Proof. Assume by contradiction that there are two correct nodes i and j which commit, respectively, r and r' with the same sequence number sn, such that $r \neq r'$. Without loss of generality, assume i commits r with sn before j commits r' with sn (according to a global clock not accessible to nodes), and let i (resp. j) be the first correct node that commits r (resp. r') with sn.

By the way we compute request sequence numbers, the fact that i and j commit different requests at the same (request) sequence number implies they commit different batches with same (batch) sequence number. Denote these different batches by B and B', respectively, and the batch sequence number by bsn.

We distinguish several cases depending on the mechanism by which i (resp. j) commits B (resp. B'). Namely, in Mir, i can commit req contained in batch B in one of the following ways (commit possibilities (CP)):

- CP1: by receiving a quorum $(n - f)$ of matching COMMIT messages in the common case of an epoch for a fresh batch B (a fresh batch here is a batch for which a leader sends a PRE-PREPARE message — see Sec. 4.3 and Sec. 4.5.3),
- CP2: by receiving a quorum $(n - f)$ of matching COMMIT messages following an ungracious epoch change, where NEW-EPOCH message contains B (Sec. 4.5.4),
- CP3: via the state transfer subprotocol (Sec. 4.5.7).

As i is the first correct node to commit request r with sn (and therefore batch B with bsn), it is straightforward to see that i cannot commit B via state transfer (CP3). Hence, i commits B by CP1 or CP2.

We now distinguish several cases depending on CP by which j commits B'. In case j commits B' by CP1 or CP2, since Mir common case follows the PBFT common case, and Mir ungracious epoch change follows PBFT view change — a violation of Agreement in Mir implies a violation of Total Order in PBFT, a contradiction.

The last possibility is that j commits B' by CP3 (state transfer). Since j is the first correct node to commit B' with bsn, j commits B' after a state transfer from a Byzantine node. However, since (1) Mir CHECKPOINT messages (see Sec. 4.5.5) which are the basis for stable checkpoints and state transfer (Sec. 4.5.7) are signed, and (2) stable checkpoints contain signatures of $2f + 1$ nodes including at least $f + 1$ correct nodes, j is not the first correct node to commit B' with bsn, a contradiction. □

4.7.3 No Duplication (P3)

(P3) **No duplication:** If a correct node commits request r with sequence numbers sn and sn', then $sn = sn'$.

Proof. No-duplication stems from the way Mir prevents duplicate pre-prepares (condition (5) in accepting PRE-PREPARE, as detailed in Sec. 4.5.3).

Assume by contradiction that two identical requests req and req' exist such that $req = req'$ and correct node j commits req (resp. req') with sequence number sn (resp. sn') such that $sn \neq sn'$.

Then, we distinguish the following exhaustive cases:

- *(i)* req and req' are both delivered in the same batch, and
- *(ii)* req and req' are delivered in different batches.

In case *(i)*, assume without loss of generality that req precedes req' in the same batch. Then, by condition (5) for validating a PRE-PREPARE (Sec. 4.5.3), no correct node preprepares req' and all correct nodes discard the batch which hence cannot be committed, a contradiction.

In case *(ii)* denote the batch which contains req by B and the batch which contains req' by B'. Denote the set of at least $n - f \geq 2f + 1$ nodes that prepare batch B by S and the set of at least $n - f \geq 2f + 1$ that prepare batch B' by S'. Sets S and S' intersect in at least $n - 2f \geq f + 1$ nodes out of which at least one is correct, say node i. Assume without loss of generality that i preprepares B before B'. Then, the following argument holds irrespective of whether i commits batch B before B', or vice versa: as access to datastructure responsible for implementing condition (5) is synchronized with per-bucket locks (Sec. 4.5.10) and since req and req' both belong to the same bucket as their hashes are identical, i cannot preprepare req' and hence cannot prepare batch B' which cannot be delivered, a contradiction. □

It is easy to see that signature verification sharding optimization does not impact the No-Duplication property. □

4.7.4 Totality (P4)

Lemma 3. *If a correct node commits a sequence number sn, then every correct node eventually commits sn.*

Proof. Assume, by contradiction, that a correct node j never commits any request with sn. We distinguish 2 cases:

1. *sn* becomes a part of a stable checkpoint of a correct node k. In this case, after GST, j eventually enters the state transfer protocol similar to that of PBFT, transfers the missing batches from k, while getting batch hash confirmations from f additional nodes that signed the stable checkpoint *sn* belongs to (state transfer is outlined in Sec. 4.5.7), a contradiction.

2. *sn* never becomes a part of a stable checkpoint. Then, the start of the watermark window will never advance past *sn*, and all correct nodes, at latest when exhausting the current watermark window, will start infinitely many ungracious epoch changes without any of them committing any requests. Correct nodes will always eventually exhaust the watermark window, since even in the absence of new client requests, correct leaders periodically propose empty requests (see Section 4.6). Infinitely many ungracious epoch changes without committing any requests, however, is a contradiction to PBFT liveness.

□

(P4) **Totality:** If a correct node commits request r, then every correct node eventually commits r.

Proof. Let i be a correct node that commits r with sequence number *sn*. Then, by (P2) Agreement, no correct node can commit another $r' \neq r$ with sequence number *sn*. Therefore, all other correct nodes will either commit r with *sn* or never commit *sn*. The latter is a contradiction to Lemma 3, since i committed some request with *sn*, all correct nodes commit some request with *sn*. Totality follows. □

4.7.5 Liveness (P5)

We first prove a number of auxiliary lemmas and then prove liveness. Line numbers refer to Mir pseudocode (Sec. 4.6).

Lemma 4. *In an execution with a finite number of epochs, the last epoch e_{last} is a stable epoch.*

Proof. Assume by contradiction that e_{last} is not stable, this implies either:

1. a gracious epoch change from e_{last} at some correct node and hence, e_{last} is not the last, a contradiction; or

2. ungracious epoch change from e_{last} never completes — since Mir ungracious epoch change protocol follows PBFT view change protocol, this implies liveness violation in PBFT, a contradiction.

□

Lemma 5. *If a correct client broadcasts request r, then every correct node eventually receives r and puts it in the respective bucket queue.*

Proof. This holds by assumption of a synchronous system after GST, and by the correct client sending and periodically re-sending request to all nodes until a request is committed (see Section 4.5.1). □

Lemma 6. *In an execution with a finite number of epochs, any request a correct node has received which no correct node commits before the last epoch e_{last}, is eventually committed by some correct node.*

Proof. Assume that a correct node i has received some request r that is never committed by some correct node j. Since (by Lemma 4) e_{last} is an (infinite) stable epoch, i is the leader in e_{last} and i will propose infinitely often exhausting all available requests until r is the oldest uncommitted request. Next time i proposes a batch from r's bucket (this eventually occurs due to bucket rotation in a stable epoch), if no other node proposed a batch with r, i includes r in its proposed batch with some sequence number *sn*.

Assume some node j, also a leader in e_{last} (since all nodes are leaders in a stable epoch), never commits any batch with sequence number sn. Then, epoch change timeout fires at j and j does not propose more batches. Eventually, epoch change timeout fires at all other correct nodes causing an epoch change. A contradiction (that j never commits any batch with sequence number sn).

Therefore, j commits a batch with sequence number sn. By (P2) Agreement the batch j commits is the same as the batch i commits and contains req. A contradiction (that j never commits req). □

Lemma 7 (Liveness with Finitely Many Epochs)**.** *In an execution with a finite number of epochs, if a correct client broadcasts request r, then some correct node eventually commits r.*

Proof. By Lemma 4, e_{last} is a stable epoch. There are 2 exhaustive possibilities.

1. Some correct node commits r in epoch preceding e_{last}.
2. r is not committed by any correct node before e_{last}. By Lemma 5 all correct nodes eventually receive request r and by Lemma 6 some correct node commits r. The lemma follows.

□

Definition 5 (Preferred request)**.** *Request r is called* preferred request *in epoch e, if r is the oldest request pending in buckets of primary of epoch e, before the primary proposes its first batch in e.*

Lemma 8. *If, after GST, all correct nodes start executing the common-case protocol in a non-stable epoch e before time t, then there exists a Δ, such that if a correct leader proposes a request r before time t and no correct node enters a view change before $t + \Delta$, every correct node commits r.*

Proof. Let δ be the upper bound on the message delay after GST and let a correct leader propose a request r before t. By the common-case algorithm without SVS (there is no SVS in a non-stable epoch e) all correct nodes receive at least $2f + 1$ COMMIT messages for r before $t + 3\delta$ (time needed to transmit PRE-PREPARE, PREPARE and COMMIT). All correct nodes will accept these messages, since they all enter epoch e by time t. As every correct node receives at least $2f + 1$ COMMITs, every correct node commits r by $t + 3\delta + rd$. Therefore, $\Delta = 3\delta + rd$. □

Lemma 9. *If all correct nodes perform an ungracious epoch change from e to $e + 1$ and the primary of $e + 1$ is correct, then all correct nodes reliably deliver the epoch configuration of $e + 1$.*

Proof. Let p be the correct primary of $e + 1$. As p is correct, by the premise, p participated in the ungracious epoch change subprotocol. Since the PBFT view change protocol is part of the ungracious view change, p sends a NEW-EPOCH message to all nodes. By the algorithm (line 157), p reliably broadcasts the epoch configuration of $e + 1$. Since all correct nodes participate in the ungracious view change, all correct nodes enter epoch $e + 1$. By the properties of reliable broadcast, all correct nodes deliver the epoch configuration in $e + 1$ (line 159). □

Lemma 10. *There exists a time after GST, such that if each correct node reliably delivers the configuration of epoch e after entering e through an ungracious epoch change, and the primary of e is correct, then all correct nodes commit e's preferred request r.*

Proof. Let p be the primary of epoch e, and C the epoch configuration p broadcasts for e. By the algorithm, the leader set C does not contain all nodes (and thus SVS is disabled in e), as all correct nodes entered e ungraciously. Since (by the premise) all correct nodes deliver C, all correct nodes will start participating in the common-case agreement protocol in epoch e. Let t_f and t_l be the time when, respectively, the first and last correct node does so.

By the algorithm, p proposes r immediately when entering epoch e, and thus at latest at t_l. Then, by Lemma 8, there exists a Δ such that all correct nodes commit r if none of them initiates a view change before $t_l + \Delta$.

Eventually, after finitely many ungracious epoch changes, where all correct nodes double their epoch change timeout values (as done in PBFT [CL02]), all correct nodes' epoch change timeout will be greater than $(t_l - t_f) + \Delta$. Then, even if a node i enters epoch e and immediately starts its timer at t_f, i will not enter view change before $t_l + \Delta$ and thus all correct nodes will deliver r in epoch e. □

Lemma 11. *There exists a time after GST, such that if all correct nodes perform an ungracious epoch change from e to $e+1$, and the primary of $e+1$ is correct, then some correct node commits preferred request in $e+1$.*

Proof. Follows from Lemmas 9 and 10. □

Lemma 12. *In an execution with infinitely many epochs there exist an infinite number of pairs of consecutive epochs with correct primaries.*

Proof. Epoch primaries succeed each other in a round-robin way across all the lexicographically ordered nodes of the system (see Sec. 4.4 and Sec. 4.2). Assume such pair of two consecutive epochs with correct primaries never exists after some epoch e. Then, in every full rotation across all $3f + 1$ nodes after e, there exist an epoch with a faulty primary node between every two epochs with correct primaries, which implies the number of faulty nodes to be greater than f, a contradiction. □

Lemma 13. *There exists a time after GST, such that for any pair of consecutive epochs e and $e+1$ with correct primaries i and j (respectively), some correct node commits at least one of the preferred requests in e and $e+1$.*

Proof. Let r_e (resp. r'_e) be preferred request in e (resp. e'). For the epoch change from e to $e+1$ there are two exhaustive possibilities.

1. *At least one correct node performs a gracious epoch change from e to $e+1$.* Recall that Mir requires the primary of an epoch to be in the leader set (Section 4.4). As e graciously ends at at least one correct node, it follows from the specification of the gracious epoch change (Section 4.5), that at least one node commits all requests proposed in e.

   Since, by the protocol, the primary of e is in the leader set of e and the correct primary always proposes the preferred request, at least one correct node commits the preferred request of e.

2. *No node performs a gracious epoch change.* This holds by Lemma 11.

□

(P5) **Liveness:** If a correct client broadcasts request r, then some correct node eventually commits r.

Proof. We distinguish two cases:

1. In an execution with a finite number of epochs, Liveness follows from Lemma 7.

2. Consider now an execution with an infinite number of epochs. By Lemma 5, every correct node eventually receives r. Let P be the set of all requests that some correct node received before it received r. After r has been received by all correct nodes, following from Definition 5, if $r' \neq r$ is a preferred request, then $r' \in P$. By Lemma 13, however, all such requests r' will eventually be committed by all correct nodes. Therefore, by Definition 5, unless r is committed earlier by some correct node, r will eventually become the preferred request of all epochs with correct primaries, and will be committed by some correct node by Lemma 13.

□

4.8 LTO: Optimization for large requests

When the system is network-bound (e.g., with large requests, such as those found in Hyperledger Fabric and/or on a WAN) the maximum throughput is driven by the amount of data each leader can send in a PRE-PREPARE message. However, data, i.e., request payload, is not critical for total order, as the nodes can establish total order on request hashes. While in many blockchain systems all nodes need data [Bit, Eth], in others [ABB+18a], ordering is separated from request execution and full payload replication across ordering nodes is unnecessary.

For such systems, Mir optionally boosts throughput using what we call *Light Total Order (LTO)* broadcast. LTO is defined in the same way as TO broadcast (Sec. 4.2) except that LTO requires property $P4$ (Totality) to hold for the hash of the request $H(r)$ (instead for request r).

label=P4 **Totality:** If a correct node commits a request r or a request hash $H(r)$, then every correct node eventually commits $H(r)$.

LTO follows SVS high-level approach (Sec. 4.5.6) and applies to Mir only in stable epoch. A leader only sends a full PRE-PREPARE message to a subset of $f+1$ *replica* nodes. To the remaining $2f$ *observer* nodes, the leader sends a lightweight PRE-PREPARE message where request payloads are replaced with their hashes. Since inside the Mir (and PBFT) common-case (Sec. 4.5.3) subprotocol, before sending a COMMIT message, a node waits to receive $2f+1$ PREPARE messages, this implies that at least $f+1$ of PRE-PREPARE messages were sent by replica nodes, ensuring that at least one correct (replica) node has the full payload.

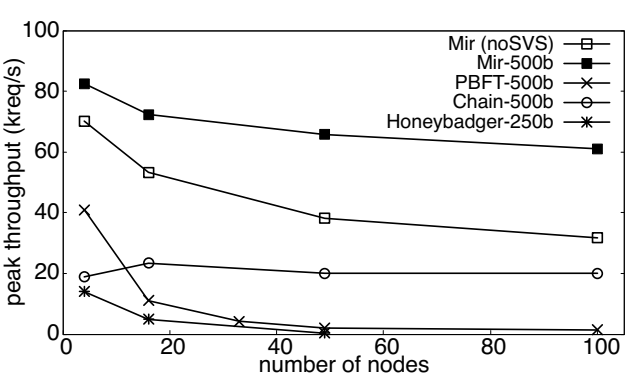
LTO has minor impact on PBFT view change (Mir epoch change) as a new primary might have a hash of the batch (lightweight PRE-PREPARE) but not the full batch payload. To this end, our Mir-LTO makes the primary in this situation look for the payload at $f+1$ replicas, which is guaranteed not to block liveness after GST and with the correct primary.

Max batch size	2 MB (4000 requests)
Cut batch timeout	500 ms ($n < 49$), $1s$($n = 49$), $2s$($n = 100$)
Max batches ephemeral epoch	256 ($n \leq 16$), $16 * n$ ($n > 16$)
Bucket rotation period	256 ($n \leq 16$), $16 * n$ ($n > 16$)
Buckets per leader (m)	2
Checkpoint period	128
Watermark window size	256
Parallel gRPC connections	5 ($n = 4$), 3 ($n = 10$), 1 ($n > 10$)
Client signatures	256-bit ECDSA

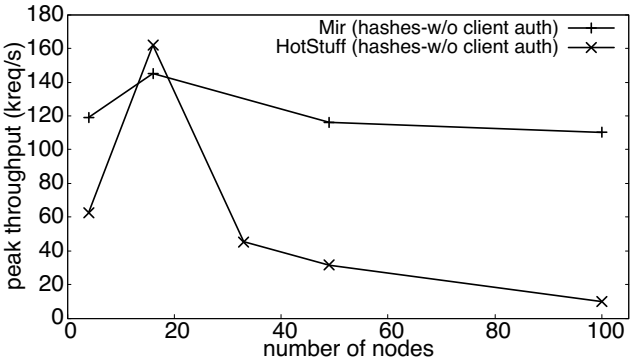
Table 4.3: Mir configuration parameters used in evaluation

4.9 Evaluation

In this section, we report on experiments we conducted in scope of Mir performance evaluation, which aims at answering the following questions:
(Sec. 4.9.1) How does Mir scale on a WAN?
(Sec. 4.9.2) How does Mir scale in clusters?
(Sec. 4.9.3) What is the impact of optimizations (SVS, LTO) and bucket rotation and what are typical latencies of Mir?
(Sec. 4.9.4) What is the benefit of Mir duplication prevention?
(Sec. 4.9.5) How does Mir perform under faults and attacks (crash faults, censoring attacks, straggler



(a) Mir vs. Chain, PBFT, Honeybadger.



(b) Mir vs. Hotstuff (no client authentication, only hashes ordered).

Figure 4.4: WAN scalability experiments.

attacks)?

Experimental Setup. Our evaluation consists of microbenchmarks of 500 byte requests, which correspond to average Bitcoin tx size [Bit19]. These are representative of Mir performance, both absolute and relative to state of the art. We also evaluate Mir in WAN for larger 3500 byte requests, typical in Hyperledger Fabric [ABB+18a] to better showcase the impact of available bandwidth on Mir.

We generate client requests by increasing the number of client processes and the request rate per client process, until the throughput is saturated. We report the throughput just below saturation. The client processes estimate which node i has an active bucket for each of their requests and initially send each request only to nodes $i-1, \cdots, i+k$, where $k \leq f-1$, i.e.,. to $f+1$ nodes.

We compare Mir to a state-of-the-art PBFT implementation optimized for multi-cores [BDK15]. For fair comparison, we use the Mir codebase tuned to closely follow the PBFT implementation of [BDK15] hardened to implement Aardvark [CWA+09]. As another baseline, we compare the common case performance of Chain, an optimistic subprotocol of the Aliph BFT protocol [AGK+15] with linear common-case message complexity, which is known to be near throughput-optimal in clusters, although it is not robust and needs to be abandoned in case of faults [AGK+15]. In this sense, Chain is not a competitor to Mir, but rather an upper bound on performance in a cluster. PBFT and Chain are always given the best possible setups, i.e., PBFT leader is always placed in a node that has most effective bandwidth and Chain spans the path with the smallest latency. We further compare to HotStuff [YMR+19] (a recent, popular, $O(n)$ common-case message complexity BFT protocol) and Honeybadger [MXC+16] using their open source implementations [6]. We present comparison to HotStuff separately, due to its implementation specifics. We allow Honeybadger an advantage with using 250 byte requests, as its open source implementation is fixed to this request size. We do not compare to unavailable (e.g., Hashgraph [L. 16], Red Belly [CNG18]) or unmaintained (BFT-Mencius[7] [MBS13]) protocols, those faithfully approximated by PBFT (e.g., BFT-SMaRt [BSA14], Spinning [VCBL09], Tendermint [Ten]), or those that report considerably worse performance than Mir (e.g., Algorand [GHM+17]).

We use virtual machines on IBM Cloud, with 32 x 2.0 GHz VCPUs and 32GB RAM, equipped with 1Gbps networking and limited to that value for experiment repeatability, due to non-uniform bandwidth overprovisioning we sometimes experienced. Table 4.3 shows the used Mir configuration parameters.

[6] https://github.com/hot-stuff/libhotstuff at commit 978f39f... and https://github.com/initc3/HoneyBadgerBFT-Python

[7] We, however, demonstrate the expected effective throughput of Hashgraph, Red-Belly and BFT-Mencius under request duplication, by "switching off" request duplication prevention in Mir, see Sec. 4.9.4.

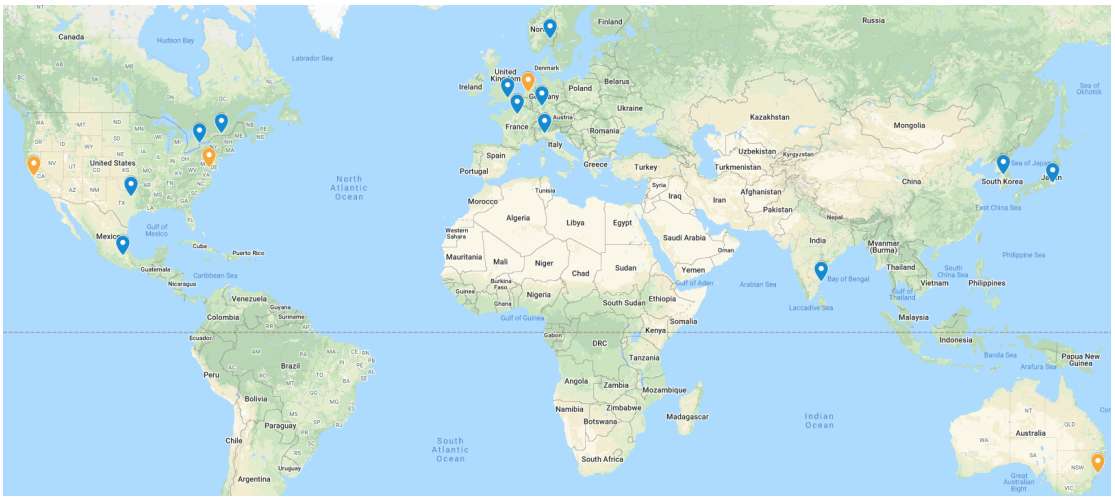


Figure 4.5: Distribution of the 16 datacenters for WAN deployment. Yellow pins indicate the $n = 4$ deployment.

Unless stated otherwise, Mir uses signature sharding optimization.

4.9.1 Scalability on a WAN

To evaluate Mir scalability, we ran it with up to $n = 100$ nodes on a WAN setup spanning 16 distinct datacenters across Europe, America, Australia, and Asia Beyond $n = 16$, we collocate nodes across already used datacenters. Our 4-node experiments spread over all 4 mentioned continents. Client machines are also uniformly distributed across the 16 datacenters. Figure 4.5 shows the datacenter distribution.

Figure 4.4a depicts the common-case (failure-free) stable epoch performance of Mir, compared to that of PBFT and Chain and Honeybadger. We observe that PBFT throughput decays rapidly, following an $O(1/n)$ function and scales very poorly. Chain scales better and even improves with up to $n = 16$ nodes, sustaining 20k req/s, but is limited by the bandwidth of the "weakest link", i.e., a TCP connection with lowest bandwidth across all links between consecutive nodes. Compared to Honeybadger, Mir retains much higher throughput, even though: (i) Honeybadger request size is smaller (250 bytes vs 500 bytes), and (ii) Honeybadger batches are significantly larger (up to 500K requests in our evaluation). This is due to the fact that Honeybadger is computationally bound by $O(n^2)$ threshold signatures verification and on top of that the verification of the signatures is done sequentially. Honeybadger's throughput also suffers from request duplication (on average 1/3 duplicate requests per batch), since the nodes choose the requests they add in their batches at random. Moreover, we report on Honeybadger latency, which is in the order of minutes (partly due to the large number of requests per batch and partly due to heavy computation), significantly higher than that of Mir. In our evaluation we could not increase the batch size as much as in the evaluation in [MXC+16], especially with increasing the number of nodes beyond 16, due to memory exhaustion issues. Finally, in our evaluation PBFT outperforms Honeybadger (unlike in [MXC+16]), as our implementation of PBFT leverages the parallelism of Mir codebase.

Mir dominates other protocols, delivering 82.5k (roughly 4x the throughput of Chain) with $n = 4$. With $n = 100$, Mir maintains more than 60k req/s (3x Chain throughput). Even without the signature verification sharding otptimization ("Mir (noSVS)") Mir significantly outperforms other protocol, delivering with $n = 4$ 70.2k req/s (3.5x Chain throughput) while reaching 31.7k req/s with $n = 100$ (1.5x Chain throughput).

Comparison to HotStuff in WANs. We present the comparison of Mir to the HotStuff [YMR+19] leader-based protocol separately, in Figure 4.4b. Despite HotStuff specifying that the leader disseminates the request payload [YMR+19], the available HotStuff implementation orders only hashes of requests, relying optimistically on clients for payload dissemination. This approach is vulnerable to liveness/performance attacks from malicious clients which can be easily mounted by clients not sending the requests to all nodes (an attack which the HotStuff version we evaluated does not address). Besides, the evaluated HotStuff implementation did not authenticate clients at all (which jeopardizes Validity).

For these reasons and for a fair comparison, we perform an experiment with: 1) disabled Mir client authentication (i.e,. client signature verification) and 2) with leaders disseminating payload hashes (relying on clients to disseminate payload as in HotStuff). We also increase batch sizes in HotStuff as much as needed, resulting in up to 32K requests per batch, to saturate the system.

We observe that HotStuff offers about 2x lower throughput than Mir with $n = 4$ nodes bounded by the number of available network connections, whereas Mir uses multiple connections among pairs of nodes. As n and number of network connections from the leader grow, HotStuff throughput first grows until the network at the leader is saturated (with $n = 16$ HotStuff performs about 10% better than Mir). However, as leader bandwidth becomes the bottleneck even with hash-only ordering, HotStuff's $O(n^{-1})$ network-bound scalability starts to show with $n > 16$, while Mir continues to scale well and is only computationally bounded by the implementation. With 100 nodes, Mir orders 110k hashes per second, compared to roughly 10k hashes per second throughput of HotStuff.

Experiments with 3500-byte payload With request payload size (500 bytes), CPU related to signature verification is the primary bottleneck. It is therefore interesting to evaluate the impact on performance with larger requests. Intuitively, with larger requests, we would be able to stress the 1Gbps WAN bottleneck of our evaluation testbed. Moreover, large requests are not only of theoretical importance, some prominent blockchain systems feature relatively large transaction sizes. For instance, minimum size transaction in Hyperledger Fabric is about 3.5kbytes [ABB+18a].

Therefore, we conducted additional WAN experiments with 3500 bytes request size.

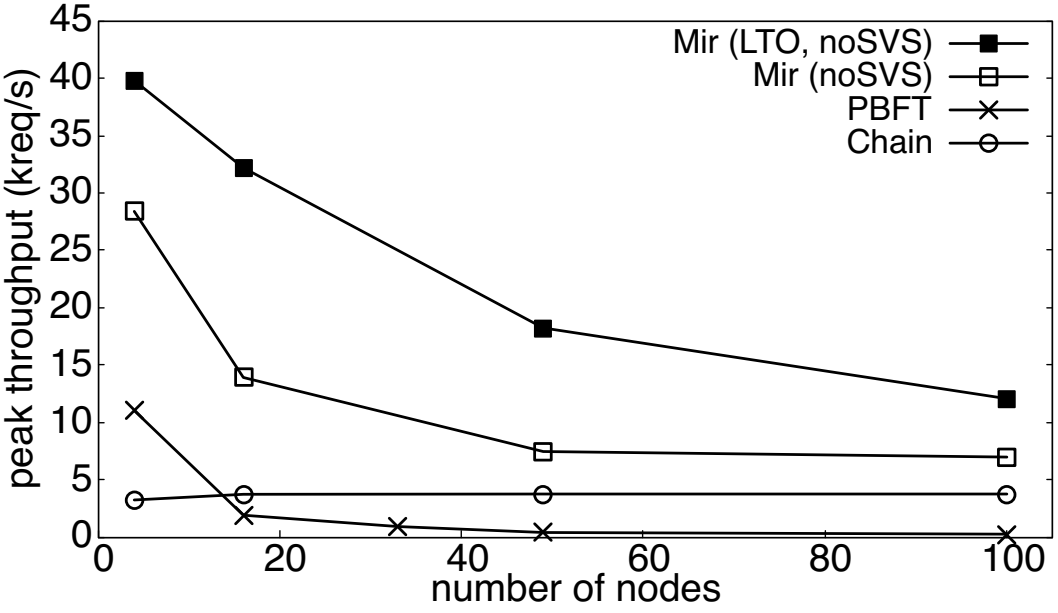


Figure 4.6: WAN scalability experiment with large payload (3500 bytes).

For large requests, where network bandwidth is the bottleneck, throughput of Mir (with no SVS) reduces to 7k req/s with 100 nodes, with a drop from 28.3k req/s with $n = 4$ nodes, see Figure 4.6. We

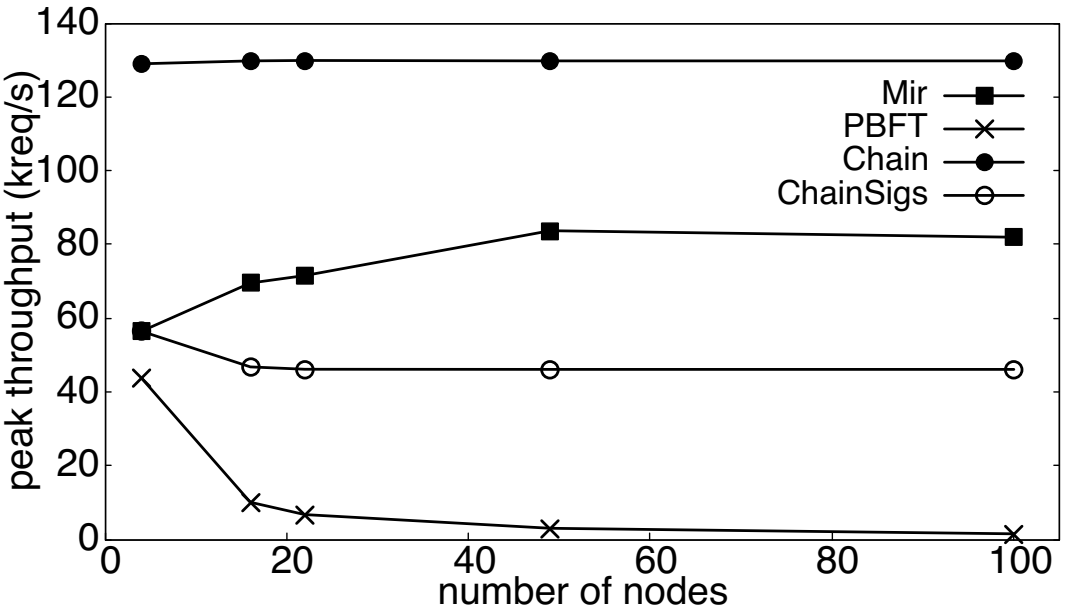


Figure 4.7: Throughput performance of Mir compared to Chain and PBFT in a single datacenter.

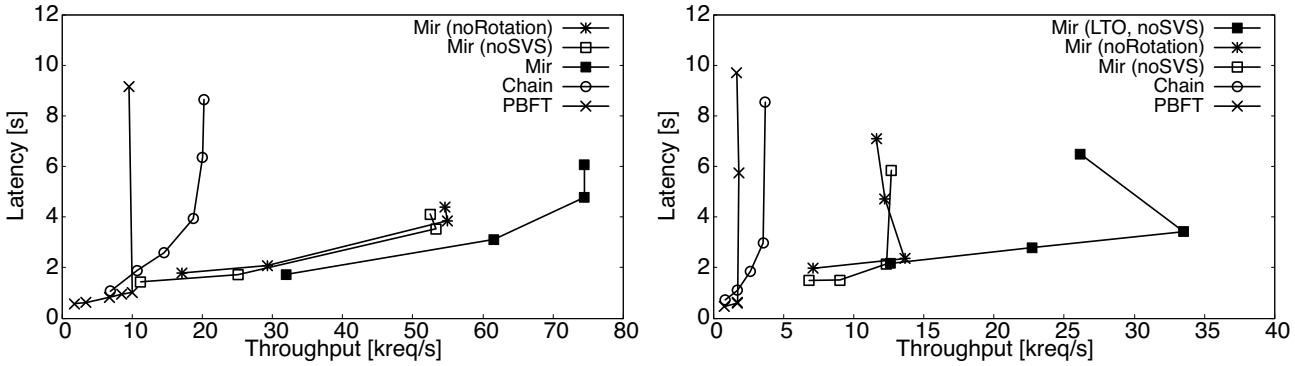
attribute this in part to the heterogeneity of VMs across datacenters (despite the identical specifications) and, most importantly, to the non-uniform partition of the available uplink bandwidth. Nevertheless, Mir delivers the best performance of all protocols to date with 100 nodes on a WAN, even compared to very optimistic protocols such as Chain, which delivers consistent throughput of about 4.5k req/s regardless of number of nodes. Mir is, hence, the first robust BFT protocol which could be used as an ordering service in Fabric with $n = 100$ nodes, without making ordering service a bottleneck (validation in Fabric is currently capped at less than 4k transactions per second [ABB+18a]).

In addition to Mir (with no SVS), Chain and PBFT, Figure 4.6 also shows an experimental variant of Mir which implements what we call *Light Total Order (LTO)* broadcast, instead of full TOB (labelled "Mir (LTO, noSVS)"). LTO is an optimization, counterpart of SVS, to help alleviate network bottlenecks in TOB. In short, LTO broadcast is identical to TOB, except that it provides partial data availability guaranteeing the delivery of the *payload* of every request to *at least one* correct node. This entails replicating batch payload to $f + 1$ nodes in stable epoch, compared to all nodes without LTO. Other correct nodes get and agree on the order of cryptographic hashes of requests, which is the basis for maintaining other TOB properties. We provide more insight into LTO in Section 4.8.

LTO boosts throughput of Mir to 40k 3500-byte req/s with $n = 4$ nodes (roughly 40% throughput improvement over Mir) and maintains about 12.5k req/s with $n = 100$ nodes (70% throughput improvement over Mir).

4.9.2 Scalability in a Cluster/Datacenter

Figure 4.7 depicts fault-free performance in a single datacenter with up to $n = 100$ nodes. Chain (the highest throughput BFT protocol to date in clusters) outperforms Mir, delivering roughly 1.6x of Mir's peak throughput (130k req/s vs 83k req/s). This difference is due to difference in client authentication: Mir verifies clients' signatures, whereas Chain uses vectors of MACs to authenticate a request to $f + 1$ replicas (these are vulnerable to "faulty client" attacks [CWA+09]). Indeed, as soon as we add clients' signatures to Chain as in robust version of Chain [AGK+15] (denoted by ChainSigs in Fig. 4.7), Chain's throughput drops below that of Mir. Mir maintains more than 80k req/s throughput, significantly outperforming PBFT.



(a) 500 byte payload

(b) 3500 byte payload

Figure 4.8: Impact of bucket rotation and Mir optimizations on a WAN with n=16 nodes.

4.9.3 Impact of optimizations and bucket rotation.

Fig. 4.8 shows the average latency and throughput of different flavors of Mir in fault-free executions using $n = 16$ nodes. We also show the performance of Chain and PBFT as a reference. Nodes are distributed over 16 distinct datacenters across the world.

Mir without signature sharding ("Mir (noSVS)" in Fig. 4.8) saturates at roughly 53k req/s (resp. 12.3k req/s for large requests), an approximate overhead of about 3% (resp. 9.5%) compared to an idealized non-robust version of Mir which involves no bucket rotation ("Mir (noRotation)"). Hence, the penalty of robust bucket rotation in Mir is small. It is more than compensated for by signature sharding which boosts Mir throughput to 74k req/s (resp. to 33.5k req/s with LTO).

All variants of Mir maintain roughly from 1–2s latency at relatively low load, to 3–5s latency close to saturation. PBFT latency is lower at 600–800 ms, yet PBFT saturates under very low load compared to Mir. We measured latency by: (1) synchronizing clocks between a client and a node belonging to the same datacenter with NTP, (2) deducting request timestamp at a client from commit timestamp at a node, (3) averaging across all requests (and, consequently, all datacenters).

4.9.4 Benefits of Duplication Prevention

In this section we examine the impact of duplicate requests to *goodput*, i.e., throughput of unique requests. In Fig. 4.9 we compare the performance of Mir (noSVS) to a version of Mir where the leaders do not partition requests in buckets, but rather add in batches all their available requests, following what Hashgraph [L. 16], Red Belly [CNG18], and BFT-Mencius [MBS13] parallel leader protocols do.

We examine the impact of duplicates in two scenarios, (1) where clients submit their requests to $f+1$ nodes — intuitively, this is the minimum number of nodes to which a client must submit a request in any BFT protocol that insures liveness (due to possible censoring by f nodes), and (2) where clients submit their requests to all nodes.

The impact is a hefty performance penalty of 61% (resp. 72%) reduction in goodput compared to Mir (noSVS) in the first (resp. second) scenario on $n = 4$ nodes. This reaches as much as 97% (resp. 99%) with $n = 100$, demonstrating $O(n^{-1})$ goodput scalability in protocols with duplication.

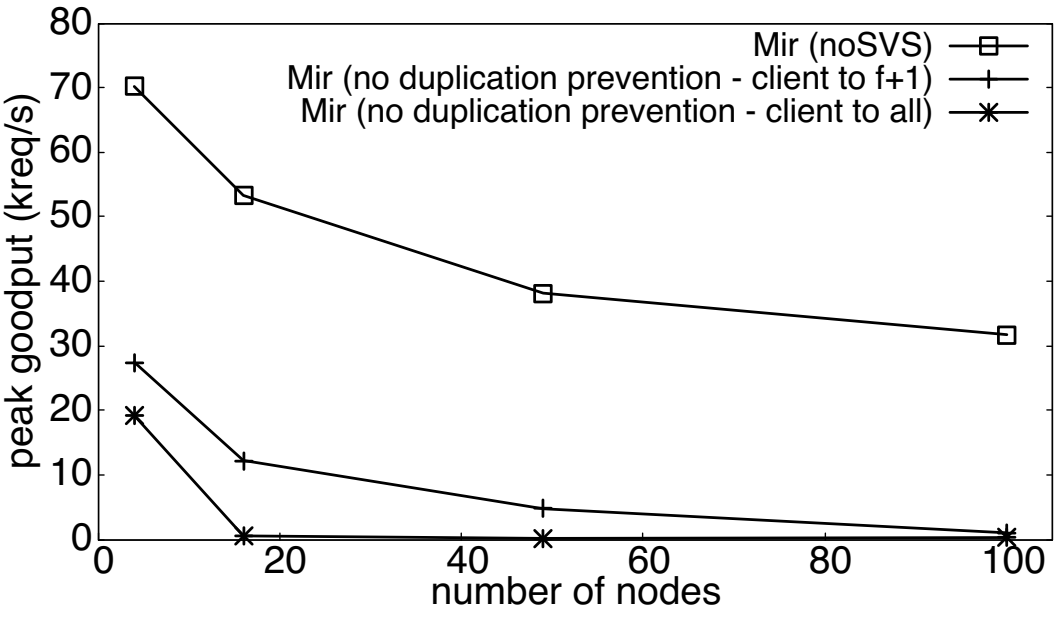


Figure 4.9: The impact of duplication prevention on a WAN with $n = 16$.

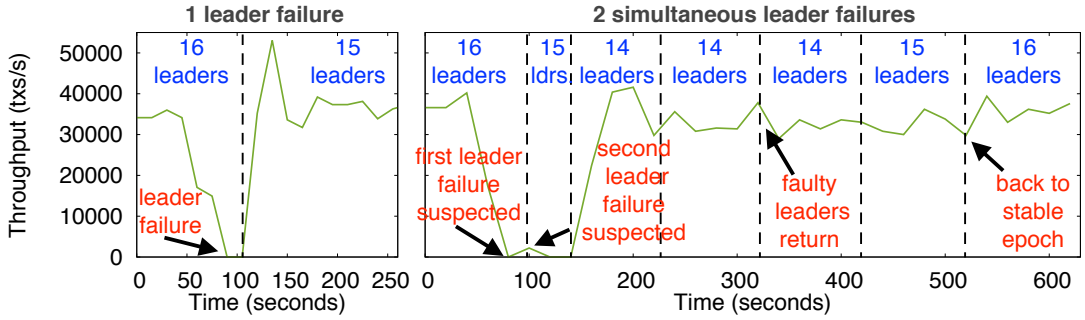


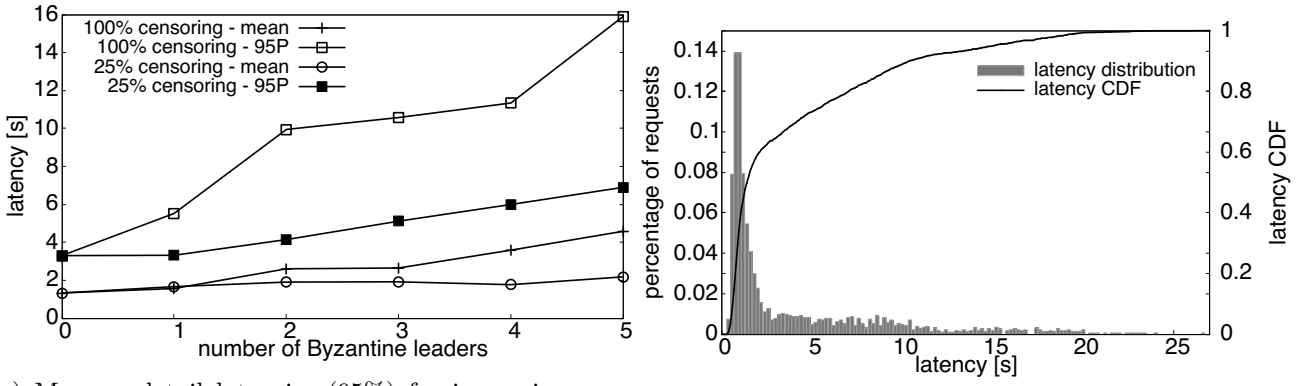
Figure 4.10: Performance under crash faults.

4.9.5 Performance Under Faults

Leader Crash Faults. Figure 4.10 shows throughput as a function of time when one and two leaders fail simultaneously. We run this experiment in a WAN setting with 16 nodes, and trigger a view change if an expected batch is not delivered with fixed timeouts of 20 seconds. With one leader failure, a view change is triggered and the system immediately transitions to a configuration with 15 leaders. When two failures occur simultaneously, the first view changes takes the system to a configuration with 15 leaders. The first few batches are delivered in this configuration, but, since one of the 15 leaders has failed, a second view change is triggered that takes the system to a configuration with 14 leaders, from which execution can continue normally. In this scenario, the figure also depicts the evolution of the leader set in case the failed nodes recover: within three epochs, the system is in a stable state with 16 leaders again.

We can observe that gracious epoch changes are seamless in Mir (these occur from second 141 onwards in the experiment with 2 faults), whereas ungracious epoch changes (when throughput temporarily drops to 0) last approximately one epoch change timeout.

Request Censoring (Byzantine Leaders Dropping Requests). In this experiment we emulate Byzantine behavior by having an increasing number (from 0 to $f = 5$) of Byzantine leaders dropping (censoring) requests in our 16-node WAN setup. Fig. 4.11a shows that mean latency remains below 4.6s



(a) Mean and tail latencies (95%) for increasing number of Byzantine leaders that drop 25% or 100% of their requests.

(b) Latency distribution and CDF with $f = 5$ Byzantine leaders censoring 100% of requests.

Figure 4.11: Latency under the request censoring attack.

(resp. 2.2s) when Byzantine leader drops 100% (resp. 25%) of the requests they receive. Tail latencies (95th percentile) remain below 16s (resp. 7s). Fig. 4.11b shows distribution and CDF of latency with 5 Byzantine leaders censoring 100% of requests. When clients sent to all nodes we observe a drop up to 15% for mean and 18% for tail latencies. A decrease of up to 44% for mean and 49% for tail latencies we observe by reducing bucket rotation period to 128 batches. This introduces, though, a trade-off of approximately 10% in peak throughput.

Stragglers (Byzantine Leaders Delaying Proposals). In this experiment we evaluate Mir resistance to stragglers. Stragglers delay the batches they lead and propose empty batches. The key to Mir straggler resistance is that a correct node starts an epoch change timeout for sequence number sn as soon as it commits $sn - 1$. With multiple leaders proposing and committing batches independently, a straggler can only impose a delay of one epoch change timeout *once per epoch* without being detected, as compared to once per sequence number in single-leader protocols.

We perform both WAN and LAN experiments with $n = 16$ nodes, starting from a stable epoch. The load is set at about 25-30% peak throughput (corresponding to roughly 25k req/s). Epoch change timeout is set to 20s and ephemeral epoch length to 256 batches. We run our experiment until the straggler is removed from and re-added to the leader set (when it becomes epoch primary).

On WAN, fault-free throughput gives a baseline of 24.8k req/s. With a single Byzantine straggler leader delaying each of its batches by 15s, the average throughput is 18k req/s (penalty of 25% over the baseline). The straggler is always detected and removed from the leader set almost immediately.

On LAN, baseline throughput without faults is 28.1k req/s. For reference, Mir latency in LAN is in milliseconds. We set straggler delay to 2 seconds (while keeping epoch change timeout to, for LAN very big, 20s) to keep the straggler longer in the leader set. This time, straggler remains in the leader set for over 600 sequence numbers, after which it is removed from the leader set. In this case, we measure average throughput of 15.7k req/s in the entire execution (a penalty of 44%).

To put these numbers into perspective, a single-leader Aardvark [CWA+09] suffers a 90% performance penalty with a straggler primary on a LAN delaying batches for 10ms. We conclude that Mir has very good performance in presence of stragglers, even with simple fixed epoch change timeouts. Future optimizations of Mir Byzantine node detection are possible, following the approaches of Aardvark [CWA+09] and RBFT [AMQ13].

4.10 Related Work

The seminal PBFT [CL02] protocol sparked intensive research on BFT. PBFT itself has a single-leader network bottleneck and does not scale well with the number of nodes. Mir generalizes PBFT and removes this bottleneck with a multi-leader approach, enforcing a robust request duplication prevention. Request duplication elimination is simple in PBFT and other single-leader protocols, where this is the task of the leader.

Aardvark [CWA+09] was one of the first BFT protocols, along with [VCBL09, AAC+08, AMQ13], to point out the importance of BFT protocol *robustness*, i.e., guaranteed liveness and reasonable performance in presence of active denial of service and performance attacks. In practice, Aardvark is a hardened PBFT protocol that uses clients' signatures, regular periodic view-changes (rotating primary), and resource isolation using separate NICs for separating client-to-node from node-to-node traffic. Mir implements all of these and is thus robust in the Aardvark sense. Beyond Aardvark features, Mir is the first protocol to combine robustness with multiple leaders, preventing request duplication performance attacks, enabling Mir's excellent performance.

The first replication protocol to propose the use of multiple parallel leaders was Mencius [MJM08]. Mencius is a crash-tolerant Paxos-style [Lam98] protocol that leverages multiple leaders to reduce the latency of replication on WANs, an approach later followed by other crash-tolerant protocols (e.g., EPaxos [MAK13]). The approach was extended to the BFT context by BFT-Mencius [MBS13]. Mencius and BFT-Mencius are geared towards optimizing latency and shard clients' requests by mapping a client to a closest node. However, as a node can censor the request, a client is forced and allowed to re-transmit the request to other nodes exposing a vulnerability to request duplication attacks which BFT-Mencius does not handle. As illustrated in our evaluation (Sec. 4.9.4), malicious clients can severely impact the throughput of such a scheme, by sending their requests to multiple or all nodes. Unlike in a regular DoS attack, these clients cannot be naively declared Byzantine or rate-limited, as such request traffic may be needed by correct clients to deal with Byzantine leaders dropping requests (request censoring attack) or to optimize the latency of a BFT protocol. Unlike BFT-Mencius, Mir maps clients' requests to buckets which are then assigned to nodes, similarly to consistent hashing [KLL+97]. Mir further redistributed bucket assignment in time to enforce robustness to request censoring. Unlike Mencius, EPaxos and BFT-Mencius, Mir does not optimize for latency in the best case, paying a small price as it does not assign clients to the closest nodes. However, our experiments show that this impact is acceptable, in particular given that the blockchain is not the most latency-sensitive application.

Recent BFT protocols, proposed in the blockchain context [CNG18, L. 16], that exhibit multi-leader flavor, also do not address request duplication. Furthermore, unlike Mir, these proposals invent new BFT protocols from scratch which is a highly error-prone and tedious process [AGK+15]. In contrast, Mir follows an evolutionary rather than revolutionary design approach to a multi-leader protocol, building upon proven PBFT/Aardvark algorithmic and systems' constructs, considerably simplifying the reasoning about Mir correctness.

Two recent protocols, HotStuff [YMR+19] and SBFT [GAG+19], are leader-based protocols that improve on PBFT's quadratic common-case message complexity and require a linear ($O(n)$) number of messages in the common case. HotStuff is optimized for throughput and features $O(n)$ messages in view change as well (SBFT requires $O(n^2)$ messages in view change). While Mir approach of multiplexing PBFT instances and SBFT/HotStuff improvements over PBFT appear largely orthogonal, our experiments show that Mir multi-leader approach scales better than HotStuff, which is a single-leader protocol. Namely, even though PBFT/Mir have quadratic common-case message complexity, these messages are load balanced across n nodes, yielding $O(n)$ messages at a *bottleneck replica*, just like HotStuff/SBFT. Our experiments also showed that HotStuff retains the downside of other single-leader protocols, i.e., bottlenecks related to leader sending all proposals, yielding an infavorable $O(n^{-1})$ throughput scalability trend. An unimplemented HotStuff variant, called ChainedHotStuff [YMR+19], suggests having different leaders piggyback their batches on other protocol common-case messages. As Hotstuff has 4

common case phases, this allows up to 4 "chained" leaders in ChainedHotStuff regardless of the total number of nodes, which is less efficient than Mir which allows up to n parallel leaders. In future, it would be very interesting to combine the two approaches, $O(n)$ common case message complexity and parallel leaders, by implementing Mir variants based on HotStuff/SBFT instead of PBFT.

Optimistic BFT protocols [AGK+15, KAD+07] have been shown to be very efficient on a small scale in clusters. In particular, Aliph [AGK+15] is a combination of Chain crash-tolerant replication [vRS04] ported to BFT and backed by PBFT/Aardvark outside the optimistic case where all nodes are correct. We demonstrated that Mir holds its ground with BFT Chain in clusters and it considerably outperforms it in WANs. Nevertheless, Mir remains compatible with the modular approach to building optimistic BFT protocols of [AGK+15], where Mir can be used as a robust and high-performance backup protocol. Zyzzyva [KAD+07] is an optimistic leader-based protocol that optimizes for latency. While we chose to implement Mir based on PBFT, Mir variants based on Zyzzyva's latency-efficient communication pattern are conceivable.

Eventually synchronous BFT protocols, to which Mir belongs, circumvent the FLP consensus impossibility result [FLP85] by assuming eventual synchrony. These protocols, Mir included, guarantee safety despite asynchrony but rely on eventual synchrony to provide liveness. Alternatively, probabilistic BFT protocols such as Honeybadger [MXC+16] and BEAT [DRZ18] provide both safety and liveness (except with negligible probability) in purely asynchronous networks. By comparing Honeybadger and Mir, we showed that this comes as a tradeoff, as Mir significantly outperforms Honeybadger, even though both protocols target the same deployment setting (up to 100 nodes in a WAN). Notably, Honeybadger authors realize the importance of duplicate elimination and suggest that each leader randomly samples the requests in their pending queue. This approach would result to no duplicates on expectation. However, in practice, unless the system is deep in saturation, the pending queue does not contain significantly more requests than the next batch. Indeed, in our Honeybadger evaluation we observed that goodput (effective throughput) was roughly only 20% of the nominal throughout. BEAT suggests some optimizations over Honeybadger without significantly outperforming the former.

As blockchains brought an arms-race to BFT protocol scalability [Vuk15], many proposals focus on large, Bitcoin-like scale, with thousands or tens of thousands of nodes [GHM+17, EGSR16]. In particular, Algorand [GHM+17] is a recent BFT protocol that deals with BFT agreement in populations of thousands of nodes, by relying on a verifiable random function to select a committee in the order of hundred(s) of nodes. Algorand then runs a smaller scale agreement protocol inside a committee. We foresee Mir being a candidate for this "in-committee" protocol inside a system such as Algorand as well as in other blockchain systems that effectively restrict voting to a smaller group of nodes, as is the case in Proof of Stake proposals [BG17]. In addition, Mir is particularly interesting to permissioned blockchains, such as Hyperledger Fabric [ABB+18a].

ByzCoin [KJG+16] scales PBFT for permissionless blockchains by building PBFT atop of CoSi [STV+16], a collective signing protocol that efficiently aggregates hundreds or thousands of signatures. Moreover, it adopts ideas from PoW based Bitcoin-NG [EGSR16] to decouple transaction verification from block mining. This approach is orthogonal to that of Mir and variants of Byzcoin with Mir instead of PBFT are interesting for future work.

Stellar [LLM+19] uses SCP, a Byzantine agreement protocol with asymmetric quorums and trust assumptions targeting payment networks, which targets similar network sizes with Mir. Asymmetric quorums of SCP modify trust assumptions and the liveness guarantees of traditional BFT protocols, with [KKK19] showing liveness violation with failures of only two specific nodes in a production configuration of Stellar. We show it is possible to obtain high throughput and low latencies while maintaining the strong guarantees of BFT protocols with classical (symmetric) quorums and trust assumptions.

Finally, *sharding* protocols [LNZ+16, KJG+18] partition transaction verification into independent shards. Mir is complementary to such protocols as they either require ordering within a shard or total ordering of the shards. Monoxide [WW19] also uses sharding to increase throughput, but provides

weaker guarantees (eventual atomicity across shards). Moreover, Monoxide's scalability heavily depends on transaction payload semantics.

4.11 Conclusions

This chapter presented Mir, a high-throughput robust BFT protocol for decentralized networks. Mir is the first BFT protocol that uses multiple parallel leaders thwarting censoring attacks and preventing request duplication. In combination with reducing CPU overhead through the "signature verification sharding" optimization, this allows Mir to achieve unprecedented throughput at scale even on a wide area network, outperforming state-of-the-art protocols.

The main insight behind Mir is multiplexing multiple parallel instances of the PBFT protocol into a single totally ordered log, while preventing duplicate request proposals by partitioning the request hash space and assigning each subset to a different leader. Mir prevents request censoring attacks by periodically changing this assignment to guarantee that each request is eventually assigned to a correct leader. Being based on well understood and thoroughly scrutinized PBFT makes it is easy to reason about Mir's correctness.

Chapter 5

Removing Data from Bitcoin Transactions

Illegal content on Bitcoin. Bitcoin [Nak09] is sometimes described as a censorship-free financial platform due to the inability of governments and institutions of blocking and restricting the creation and transfer of the cryptocurrency. Unfortunately, recent research [MHH+18, BMS19] raises issues on the immutable Bitcoin's blockchain. Indeed, the Bitcoin's blockchain can also store non-financial data and among them Matzett *et al.* [MHH+18] discovered files with child pornography and links to dark web services. As a result, the researchers argued that "...it could become illegal (or even already is today) to possess the blockchain, which is required to participate in Bitcoin.".

Methods to inject illicit data on the blockchain. Henceforth, we will use the term *blockchain* as synonym for the Bitcoin's blockchain.

The most typical transactions in Bitcoin are based on the Pay to-Public Key Hash (P2PKH) mechanism which allows transfers of funds from one user to another. Other common transactions include the Pay to Public Key (P2PK) mechanism used until 2012 in coinbase transactions to remunerate miners, the multi-signature mechanism and data output (`OP_RETURN`) transactions. We defer to Bistarelli *et al.* [BMS19] for a complete analysis and statistics of standard and non-standard transactions.

In the blockchain, non-financial data can be encoded arbitrarily: the space of possible encodings is infinite. For instance, somebody could claim that a given function f applied to a set S of blocks results into a string that represents illicit content and as consequence asks for deletion of any block in S. If the function f is simple the illicit content may be evident, e.g., it can be looked up as a consecutive string of characters. However, the function f might have the following complex form: take the characters number $i_1, \ldots, i_n$ from the blockchain and concatenate them together; the resulting string might encode an offensive English sentence.

Therefore, it is not generally possible to even define where and how illicit content is stored. For this reason, we have to be concrete and focus on specific patterns and places in which illicit content can and has been stored on the blockchain so far. (In the following we assume the reader to be familiar with the Bitcoin internals.)

- Coinbase transactions. A coinbase transaction is a transaction in which the `TXID` field is all zeros and the `VOUT` field is all ones (since it does not refer to an existing transaction and does not refer to an existing output), and the `scriptSig` field can actually contain arbitrary data. For instance, the `scriptSig` field of the genesis coinbase transaction, identified by `TXID` 4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b, is (decoded as) the string "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks".

- Data output transactions. The `OP_RETURN` mechanism is used to store non-financial data and this

is usually done through the use of `OP_RETURN <DATA>` inside the script specified in a transaction. The `OP_RETURN` functionality was actually introduced in Bitcoin with the purpose of mitigating the misuse and abuse of coinbase and other mechanisms to store data on the blockchain.

5.1 The Limitations of Previous Solutions and Our Scenario

The first approach to the problem of data removal from blockchains was proposed in the work of Ateniese *et al.* [AMVA17] that mainly tackles the permissioned setting and thus has no applicability to Bitcoin.

Puddu *et al.* [PDC17] propose a protocol in which users can set alternate versions, called "mutations", of their transactions that can be later activated after running an expensive MPC protocol. A request of a modification has to be approved by means of a voting procedure based on proofs of work. In this solution, only the creator of a transaction can allow modifications, thus preventing deletion of content inserted by malicious parties.

Deuber *et al.* [DMT19] aims at constructing a publicly verifiable redactable blockchain in the permissionless setting. In the Deuber *et al.* protocol, each user can propose a modification by writing it on the blockchain. The redaction proposal is subject to a voting procedure based on consensus and computational power. The redaction is considered accepted by a majority if a certain number of blocks confirming the proposal is added to the blockchain and in such a case, the old block can be accordingly redacted. The new proposed block along with the next blocks produced during the voting procedure guarantee the public verifiability.

Thyagarajan *et al.* [TBM+20] propose Reparo, a protocol that improves on the Deuber *et al.*'s protocol by adding the possibility of redacting or modifying already (before the fork) inserted blocks in the blockchain, a property that they call *Repairability of Existing Content* (REC).

The common thread in the aforementioned approaches for the permissionless setting is the use of consensus protocols to agree on a redaction. Is consensus and agreement on redactions really needed?

Until now, we used the term "illicit content" without a definition. The categories of content that can harm the Bitcoin users and in particular the nodes keeping the blockchain is very diversified and does include material like illegal pornography, privacy and copyright violations and malwares. However, in some countries unauthorized content may be also of religious or political nature.

The *duty* of deletion of some content might be imposed by an authority to a set of nodes falling under its legislation and not to others. Yet, different legislations do or could not agree on which contents to redact.

For this reason, we envision a scenario in which a Bitcoin node keeping the full blockchain wants to or is requested to delete some string from either a coinbase or a data output transaction (of the general form described before). Therefore, our approach departs from previous ones in that we deem not reaching a consensus on which content to delete as a *feature*. We call such a property *Individual Honest Deletion* (IHD) or simply individual deletion. Moreover, consensus protocols are expensive and require a lot of time to reach the agreement.

Quality of the deletion decision. Another issue with the Deuber *et al.* and Reparo protocols, when instantiated for Bitcoin, lies in the fact that even if the adversary does not have half of the hashing power can have impact on the voting procedure. According to Garay *et al.* [GKL15a], an adversary controlling a fraction t of the hashing power can control up to a fraction $\frac{t}{1-t}$ of the blocks in the chain.

Thyagarajan *et al.* (Appendix E) concretely propose to consider a redaction in Bitcoin accepted if it received 50% (threshold $\frac{1}{2}$) of the votes in the 1024 blocks after the redaction proposal. Due to Garay *et al.*'s analysis, we deem such a 50% parameter too optimistic since it would render the voting procedure and so the ability of redacting the blockchain monopolized by a malicious attacker owning about $\frac{1}{3}$ of the hashing power.

We say that a protocol for deletion in Bitcoin achieves *t-quality* if no adversary controlling a fraction $t < \frac{1}{2}$ of the hash power can delete some arbitrary content stored in other nodes if all other nodes are against deletion. As argued above, the protocols in the aforementioned works are such that whatever threshold f is selected there exists a value $t < \frac{1}{2}$ such that an adversary controlling a fraction t of the hash power succeeds in the attack.

For instance, Reparo instantiated with voting threshold $\frac{1}{2}$ (as suggested by the authors) and assuming adversaries owning $\frac{1}{3}$ of the total hashing power does not satisfy $\frac{1}{3}$-quality neither does satisfy $\frac{2}{5}$-quality when instantiated with voting threshold $\frac{2}{3}$ and assuming adversaries controlling a fraction $\frac{2}{5}$ of the hashing power. But Reparo for threshold parameter $\frac{3}{4}$ and assuming adversaries controlling $\frac{4}{10}$ of the total hashing power satisfies $\frac{4}{10}$-quality. Notice that, in order to make Reparo achieve a certain quality, the protocol is then subject to the symmetrical issue: in the latter example, if a large majority, e.g., $\frac{3}{4} - \frac{1}{100}$, is in favour of deletion, the adversary can subvert the decision. We informally say that a protocol has good quality whether it is not subject to such issues.

Redaction protocols based on voting are also prone to bribing attacks in which a player interested in deletion can bribe other users out-of-band.

Despite the fact that we do not seek for consensus on deletion, we do require *public verifiability*, the absence of which would render the problem trivial. We remark that here, by public verifiability we mean the ability of verifying the consistency of the Bitcoin's financial state and not, like in Thyagarajan *et al.*, the accountability of redactions.

Our main questions. Having discussed the specific points on the blockchain in which illicit content can be added (and has been added so far) and the actors interested in the deletion (i.e., a Bitcoin node on an individual basis or under request from an authority), it is natural to pose the following questions:

- Question I: can illicit content be ever deleted by individual nodes without requiring a *hard* fork and preserving public verifiability? The obvious requirement is that the deletion of the illicit content should not affect the blockchain in any harmful way, e.g., by invalidating other content or the financial integrity. If a node could not delete a string without being unable to participate in Bitcoin in the future, the owner of the node would be in a lose-lose situation: (1) delete the string and shutdown the node or (2) keep on participating in Bitcoin but at the cost of infringing the laws.
- Question II: If it is not possible to delete illicit content *in general* by individual nodes, under what circumstances is it instead possible?

5.2 Our Contributions

Our results can be summarized as follows.

- We answer Question I by carefully analyzing the Bitcoin protocol and showing in which cases naive deletion may be harmful and why it is not possible *in general* to delete content from Bitcoin.
- Having clarified that trivial deletion is not possible, we provide a theoretical solution to sanitize Bitcoin from illicit content in the specific cases in which it is possible and present tweaks to the Bitcoin protocol to enable deletion.
- We demonstrate our approach to be practical by providing a tool to sanitize the Bitcoin blockchain. We explain how our tool can be used black-box to sanitize Bitcoin presenting concrete examples.

1. Take `outScript` and delete all instances of `OP_CODESEPARATOR` and denote by `Subscript` the result of this step.
2. Copy t' in string `TxCopy`.
3. Remove from `TxCopy` all sections relative to the input scripts.
4. Add to `TxCopy` the script `Subscript` in the point where at the previous step the input scripts have been removed.
5. Sign the so computed string `TxCopy`.

Figure 5.1: Bitcoin's algorithm used to sign transactions.

5.2.1 Can data be removed from Bitcoin in general?

A naive thought can suggest that deleting illicit content from data output transactions is innocuous since "strings following `OP_RETURN <DATA>`" are not necessary to keep the integrity of transaction and data output transactions cannot be redeemed; and similar consideration applies to coinbase transactions.

The issue of signatures. To elaborate on this, we firstly need to analyze how transactions are signed in Bitcoin. To sign a transaction, the following steps are carried out. Let t be a transaction that has to be redeemed and t' a transaction that redeems t. Let `outScript` the script of output in t and let `inScript` the input script in t' that, concatenated to `outScript` returns `TRUE` on the stack and renders the transaction t' valid. If in `outScript` is encountered the command `OP_CHECKSIG`, the signature is computed as shown in Figure 5.1.

The issue is that the `Script` language has logical opcodes `OP_IF`, `OP_NOTIF`, `OP_ELSE`, `OP_END` that allows an `OP_RETURN` to be set in a branch that is *never* executed. In this case a redeemable output script can also contain a substring of the form `OP_RETURN <DATA>` and, according to the previous algorithm, a redeeming input script of a subsequent transaction needs to sign the concatenation of such output script with all the input scripts in the redeeming transaction. The script in Figure 5.2 is an example. The instruction 3 will be never executed whatever input script in a future transaction will try to redeem the above output script; only instructions 5-9 will be executed as in a typical `scriptPubKey` script of P2PK transactions.

We also stress that the signature in a redeeming script is (skipping details) the signature of the string resulting from the concatenation of the redeemed output script *as it is* and the redeeming transaction after that all input scripts are removed by the redeeming transaction. Indeed, if the Bitcoin's specification happened to give as input to the signature the *hashes* of the redeemed output script, one could delete the illicit content and only save the hashes for future use. (Technically, this would still require a software update for the nodes.)

Therefore, we conclude that a hard fork is needed to enable deletion of illicit content *in general*. An objection to the conclusion could be that scripts like the one in Figure 5.2 are artificial and indeed are non-standard.

A standard script is one that passes the `isStandard()` and `isStandardTx()` tests in `policy/policy.cpp` of Bitcoin Core. However, as shown by Bistarelli *et al.* a non-negligible quantity of non-standard transactions have been mined by pools of miners that deviate from the standard Bitcoin Core implementation.

```
1. OP_TRUE
2. OP_NOTIF
3. OP_RETURN <Idiot who reads>
4. OP_ENDIF
5. OP_DUP
6. OP_HASH160
7. <pubKeyHash>
8. OP_EQUALVERIFY
9. OP_CHECKSIG
```

Figure 5.2: Script that contains a branch that is never executed in which arbitrary content can be stored.

Moreover, if any measure to mitigate the problem of illicit content were ever conditioned to the absence of non-standard transaction, more nodes could start accepting and mining non-standard transactions as an attack.

5.3 Related Work and Comparison

Ateniese *et al.* [AMVA17] propose the first protocol for illicit content deletion from blockchains. Their solution is simple and efficient but, unfortunately, mainly targets the permissioned setting and cannot be adapted to Bitcoin. Moreover, unlike ours, in their approach a deletion does not leave trace and goes unnoticed to users not participating in the redaction.

Paddu *et al.* [PDC17] provide a more complex protocol for dealing with redactions of harmful content. The main drawback of their solution is that the ability of "mutating some content" (in their terminology) has to be explicitly set by the miners and so malicious miners can simply bypass the mutation mechanism. Moreover, mutation of some content has a cascade effect on any subsequent transaction, thus incurring a huge performance penalty.

Deuber *et al.* [DMT19] propose a novel redactable blockchain protocol that can be integrated in Bitcoin. The Deuber *et al.*'s proposal requires users to interact online in a consensus protocol to request approval from the majority whereas in our protocol each node can *individually* perform a deletion without the need of interaction with another nodes. Dueber *et al.*'s introduce the property of "public verifiability", that is the ability of tracing redactions. In our protocol redactions can be traced as well. However, we notice that this form of accountability can be actually achieved in all natural protocols so we do not consider it as a fundamental property.

Thyagarajan *et al.* [TBM+20] propose Reparo, a protocol that improves the Deuber *et al.*'s solution with the property of "Repairability of Existing Content" (REC), that is the possibility of redacting or modifying already (before the fork) inserted blocks in the blockchain. As in Deuber *et al.* Reparo is based on expensive and interactive consensus protocols that requires several *days* to be run as opposed to our protocol in which deletion can be performed in few *minutes*.

Both Thyagarajan *et al.* and Deuber *et al.* do not guarantee individual deletion and *good quality* as defined and discussed in Section 5.1.

Solution	Permissionless?	REC?
Ateniese *et al.* [AMVA17]	×	×
Puddu *et al.* [PDC17]	√	×
Deuber *et al.* [DMT19]	√	×
Thyagarajan *et al.* [TBM+20]	√	√
Ours	√	√

Table 5.1: Comparison of our solution with the state of the art in redaction of blockchains.

Solution	Individual Deletion?	Good Quality?	Runs in few minutes?
Deuber *et al.* [DMT19]	×	×	×
Thyagarajan *et al.* [TBM+20]	×	×	×
Ours	√	√	√

Table 5.2: Comparison of our solution with the protocols of Deuber *et al.* and Thyagarajanan *et al.* when instantiated for Bitcoin. The individual deletion and good quality properties are defined in Section 5.1.

Florian *et al.* [FHBS19] put forward a different approach to deletion in Bitcoin in which the nodes do not completely validate the chain while in our solution a blockchain subject to deletions of data can be completely validated from any other node in the network and vice versa.

In Table 5.1 we summarize the state of the art comparing it to our solution.

In Table 5.2 we compare our work to Deuber *et al.* and Thyagarajan *et al.* when instantiated for Bitcoin.

5.4 Preliminaries

5.4.1 Bitcoin in a nutshell

Bitcoin [Nak09] is a permissionless blockchain system that allows users to perform electronic payments based on cryptographic proofs instead of trust. Two parties interested in exchanging coins can transact directly without the need of a trusted party. An electronic coin is a chain of digital transactions and each owner can transfer his coin to another party using one of the possible standard transactions:

- Pay to Public Key (P2PK), a public-key script used to send a transaction to one Bitcoin addresses;
- Pay To Public Key Hash (P2PKH), the most common form of public-key script used to send a transaction to one or multiple Bitcoin addresses;
- Pay To Script Hash (P2SH), used to send a transaction to a script hash;
- Multisig, used to require multiple signatures before a UTXO can be spent;
- Pubkey, that are a simplified form of the P2PKH public-key script.

Since Bitcoin was used to store data on the blockchain, another standard transaction was added to add arbitrary data to a provably unspendable public-key script. This transaction is called NullData transaction and uses the `OP_CODE OP_RETURN` to store arbitrary data on the blockchain. It is preferable to use NullData transactions to insert arbitrary data on Bitcoin instead of using the other transactions, since NullData transactions can be automatically pruned by the UTXO database.

All transactions are public and can be viewed by everyone that can check that the blockchain is valid and no coins are double spent.

Bitcoin is based on a Proof-of-Work system, every time that a miner Miner wants to publish a new block, Miner needs to solve a cryptographic puzzle, that consists of scanning for a value that when hashed, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash. Rewrite the Bitcoin transaction history means recompute all the blocks after the modified block.

The following operations are performed in the Bitcoin network:

- every time that a party has a new transaction tr, tr is broadcast to all nodes;
- each miner collects new transactions into a block;
- each miner works on finding a difficult proof-of-work for its block;
- when a miner finds a Proof-of-Work, it broadcasts the block to all nodes;
- nodes accept the block only if all transactions in it are valid and not already spent;
- miners express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Miners always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next Proof-of-Work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

To modify a past block, an attacker would have to redo the Proof-of-Work of the block and all blocks after it and then generate enough new blocks to surpass the work of the honest parties.

The first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a given number of coins is reached, the only incentive for miners will become the transaction fees.

In order to save disk space, it is possible to delete locally the oldest data of Bitcoin. It is possible to delete local data without breaking the block's hash, since transactions are hashed in a Merkle Tree, with only the root included in the block's hash.

A node that maintains all the Bitcoin history is a full network node. It is possible to verify payments without running a full network node. A user only needs to keep a copy of the block headers of the longest Proof-of-Work chain and obtain the Merkle branch linking the transaction to the block it is timestamped in. He cannot check the transaction for himself, but by linking it to a place in the chain, he can see that a network node has accepted it, and blocks added after it further confirm the network has accepted it.

5.4.2 SNARKs/STARKs

We assume the reader is familiar with succinct ZK proofs (SNARKs) [Mic00,Kil95,GGPR13]. SNARKS are essentially non-interactive ZK proofs that have short size and fast verification. For our solution to work, succinctness is not strictly necessary but offers a clear improvement in terms of parameters. STARKs [BBB+18b, BCS16] are variants of SNARKs that do not require trusted parameters.

5.4.3 Isekai

Our implementation is based on Isekai, a versatile framework for verifiable computation. Isekai allows to transform a C/C++ program into a set of R1CS constraints, an internal representation for many SNARK and STARK systems. Then, Isekai offers an interface to several SNARK/STARK systems like the SNARK of [GGPR13], Bulletproof [BBB+18b] and Aurora [BCS16] allowing to invoke the prover and the verifier of such system in a black-box way.

Usage. Isekai can generate a proof of the execution of a C/C++ function. The C function must have one of the following signatures:

```
void outsource(struct Input *input, struct NzikInput *nzik, struct Output *output);
void outsource(struct Input *input, struct Output *output);
void outsource(struct NzikInput *nzik, struct Output *output);
```

`Input` and `Output` are public parameters and `NzikInput` is the private input. The inputs are given in an external file with the same name of the C program but with extension `.in` by putting each value one per line. With the option `--r1cs` the R1CS files are generated from the `.in` file and then with this R1CS it is possible to generate the proof using the `--prove` option. The proof is verified using the `--verif` option. The specific SNARK/STARK scheme is chosen using the option `--scheme`.

Isekai does not offer a way to encode the secret input for the verifier in the R1CS format - this can be done only for files containing *both* public and private inputs. For these reasons, in our implementation we had to add this functionality to Isekai.

5.5 Our Bitcoin Sanitizer

We will first show that the problem of secure deletion from Bitcoin boils down to computing and verifying non-interactive zero-knowledge (NIZK) proof [DMP88] for conceptually simple (class of) statements. Then we will show how to implement proofs for such statements in an *efficient* way by losing only harmless information.

The general statement. Let h and $X_1, \ldots, X_n, X_{n+1}, n \geq 1$ be public strings (possibly empty) and let H be the SHA256 function used in Bitcoin. Consider the following statement $\mathsf{PreImage}_{h,X_1,\ldots,X_{n+1}}$:

$$\exists y_1, \ldots, y_n \; H(X_1||y_1 \cdots X_n||y_n||X_{n+1}) = h,$$

in which we implicitly assume that the indices of the substrings $y_1, \ldots, y_n$ and their length are public and part of the statement and for simplicity omitted. When it is clear from the context we will drop the subscript.

We will next show how all cases of deletion we want to take into account can be reduced to proving and verifying in zero-knowledge (ZK) the previous class of statements.

- Deletion from input scripts and non-redeemable output scripts. This can be the case of illicit content in coinbase transactions, illicit content of the type `OP_RETURN <DATA>` nested inside a branch of an input script that is never executed and standard data output transactions that are not redeemable.

  In all such cases, the transaction has the form $s = X_1||y_1|| \cdots X_n||y_n||X_{n+1}$ such that $H(s) = h$ and the substrings $y_1, \ldots, y_n$ represent illicit content. Observe that the case $n > 1$ models the possibility of having, e.g., multiple `OP_RETURN <DATA>` occurrences nested inside a script or the fact that only some parts (but not all) of the string `DATA` in a `OP_RETURN <DATA>` or some parts in a `scriptSig` of a coinbase transaction have to be deleted.

A node A proceeds as follows. The node computes a NIZK proof π of the statement $\mathsf{PreImage}_{h,X_1,\ldots,X_{n+1}}$ and then replaces all occurrences of $y_1, \ldots, y_n$ by zeros.

What is the purpose of the NIZK proof? In a normal scenario (when no deletion happens), when a node B requests from A a block B, B has to verify that B is "consistent" with all transactions inside it. Precisely B contains a field called `Merkle Root` that, basically, is the root of the Merkle tree that has the transactions as leaves. Each transaction can be hashed so that the root of the transactions can be reconstructed and compared to the field `Merkle Root`. When deletion happens, a leaf representing a transaction subject to deletion would have an invalid hash (due to the fact that the illicit content has been replaced by zeros), so the node B would have to reject the block.

Here, it is where the NIZK proof π comes into the play. By means of π, E can check that the block, identified by `Merkle Root`, is consistent with *some* set T' of transactions that is identical to the set T in B except for some substrings. (Recall that the indices in which the substrings y_i's occur and their lengths are public.)

- Deletion from output scripts that are redeemable. In this case, the issue is the following. We have an output script `out` in which a node A performs a deletion at time t. At a time $t' > t$, a node B that is not aware of the deletion generates a transaction containing, e.g., a single input script `inp` attempting to redeem `out`. Node A has to sign the string s resulting from the concatenation of `out` and `inp`. Observe that node A, as well as any other node that has downloaded the same block from A, lost the original string `out` so is *unable* to verify the signature.

  This issue is solved by tweaking the Bitcoin signature as being signature of the concatenation of the *hashes*, i.e., setting the string s to be signed as concatenation of `H(out)` and `H(inp)`. Indeed, notice that whatever string s has to be signed, the signing algorithm internally "signs" the digest `H(s)`. Therefore, we can tweak the `OP_CHECKSIG` protocol as follows. First, attempt to verify the signature as usual. If the verification algorithm fails, check a NIZK proof π of the fact the string `h=H(s)` is such that `s` is the concatenation of `H(out)` with `H(inp)` and a NIZK proof π_2 of the fact that `H(out)` is consistent with `out` after the deletion. Both statements can be expressed as a special case of the class of statements $\mathsf{PreImage}$.

Multiple deletions at different times. Our solution allows deletions of illicit content from different transactions in the same block at different times. That is, a transaction T_1 in a block D can be subject to deletion at time t_1 and later at time t_2 a transaction T_2 belonging to B can be subject to deletion as well. We note that in our solution we do not consider the case in which the same transaction T_1 has to be modified multiple times in different time.

Breaking the general statement in more "efficient" statements. We could implement our system using a SNARK proof for the class of statements $\mathsf{PreImage}_{h,X_1,\ldots,X_{n+1}}$ described above. There are two problems with this approach, one theoretical and one practical. The first problem is that, even if there is only a single and short deletion of few bytes, the complexity will depend on the length of the overall transaction and this is a wasteful overkill. The second problem is that for larger transactions, the length of the R1CS circuits generated by the Isekai system becomes huge. For transaction $>$ 1KB, we would need hundreds gigabytes.

Instead of proving and verifying the previous statements directly in ZK (i.e., using a ZK proof system for those statements), we essentially prove and verify such statements in a more efficient way. The idea is to consider all intermediate outputs of each round of SHA. Recall that SHA essentially works as follows: given an input X, it extends X to an input X' of a length multiple of 64 bytes, break X' into chunks (blocks) of 64 bytes and for each of such chunks it executes a round function $\mathsf{SHARound}$

that takes as input a chunk and the output of the previous round. The first round takes as input the first chunk and a fixed value h_0.

Let X be a string that has been redacted in some points starting from a string Y and let $h = H(Y)$. Recall that the string after redaction X and the hash h are public information as long as the points in which the redaction has been done. The secret is the original string Y before the redaction. Our scope is to design an efficient proof system to convince a verifier that the public inputs are consistent with the redaction.

Let us say that SHA extends Y (resp. X) into a string Y' (resp. X') consisting of m chunks $Y_1, \ldots, Y_m$ (resp. $X_1, \ldots, X_m$) of 64 bytes. The prover will reveal the intermediate outputs $h_1, \ldots, h_m = h$ of each round[1], where for each $1 \leq i \leq m$ $h_i = \mathsf{SHARound}(h_{i-1}, Y_i)$.

Then, *only for one of intervals subject to redaction*, the prover proves using the Aurora proof system (see below) that there exists a string X_i such that Y_i is the result of applying the redaction to X_i and $h_i = \mathsf{SHARound}(h_{i-1}, Y_i)$. The verifier verifies each ZK proof relative to the chunks subject to redaction and for each other chunk i not subject to redaction additionally verifies that $h_i = \mathsf{SHARound}(h_{i-1}, Y_i)$; notice that the latter is verified just by running the round function on the known preimage.

We can see the overall proof system as a proof system for the class of statements $\mathsf{PreImage}_{h,X_1,\ldots,X_{n+1}}$ described above. Indeed, completeness and soundness are easy to check. ZK does not hold since, revealing the intermediate outputs of the SHA function does reveal whether two transactions have a common prefix. However, we adopt a pragmatic approach and do not consider such a leakage as harmful. Moreover, we do not employ complex ZK proofs for the intervals not subject to the redaction, so preserving efficiency.

Practicality of our construction. Even though our solution seems theoretically and conceptually simple, it was never adopted or proposed before, perhaps due to the belief that ZK proofs are not practical. We remark that it is out of the scope of this chapter to provide a complete implementation of our Bitcoin sanitizer. Notwithstanding, taking advantage of recent progress on succinct ZK proofs (SNARK/STARKs) [GGPR13, BBB+18b, BCS16], we demonstrated the feasibility and practicality of our approach by providing a Bitcoin *sanitizer* that is fully functional and that can be integrated later with Bitcoin (or even other blockchains).

The purpose of our Bitcoin sanitizer is actually to give a lower-bound on the practicality of our approach, showing that our solution allows to perform redactions in *minutes* rather than *days* as for previous solutions based on different approaches. Therefore, we were driven to SNARK/STARKs combined with Isekai (see Section 5.4.3) from the need of an easy instrument to convert C++ code into ZK proofs.

Among other ZK tools our choice fell on Aurora because it is at the same time *transparent* (no need for trusted parameters) and has short proofs. Having a trusted setup in the context of deletion from Bitcoin would be questionable.

Observe that the toolkit for ZK proofs for ledgers that is part of the activities of PRIViLEDGE but according to the description in Deliverable 4.2 (chapter 5) would only partially fulfill our needs. Indeed, such toolkit implements SNARKs in the CRS model with subversion-resistance. The latter property solves the problem of the trust in the CRS with respect to the ZK property. However, the setup still needs to be trusted with respect to the soundness property. Soundness with updatable CRS goes in the direction of mitigating this issue. Unfortunately, this would require additional changes/complications at the protocol layer in order to allow sequential updates of the CRS.

[1]Notice that it is not necessary for the prover to send all intermediate output, but only the intermediate outputs of the modified chunks, since the outputs of the unmodified chunks can be independently computed by the verifier.

5.6 Our Implementation

In this section we provide a demo implementation of our Bitcoin sanitizer, a tool that can be integrated in Bitcoin to deal with deletions of illicit content from Bitcoin.

Our implementation is based on Isekai (see Section 5.4.3) as compiler to convert our C++ programs into R1CS representation. Isekai also provides an interface to several major SNARK/STARK systems and our choice fell on Aurora due to its following unique properties.

- Post-quantum. Aurora is post-quantum secure guaranteeing security even against future advances in quantum technology.
- Fast verification. Aurora does not just provide short proofs but offers a verifier of *logarithmic* time in the circuit size compared to the linear performances of other SNARK/STARK systems. Having used a system like Ligero or Bulletproof with linear verification[2] would have rendered our solution infeasible since deletion is an infrequent operation but verification is frequent.
- Transparency. Aurora is transparent meaning that there is no trusted setup like in Gentry *et al.*'s [GGPR13] SNARK. Having used a CRS-based SNARK, our solution would have introduced the annoying issue of the secure generation of the CRS.

Our implementation is deployed for the Linux OS and consists of the following modules. The statement proved by our implementation is the following. Let

- X be the original transaction padded to a multiple of 64 bytes as described by SHA256 specifications;
- $y_1, \ldots, y_m$ be the bytes to delete by X;
- Y be the transaction obtained substituting $y_1, \ldots, y_m$ from X with 0 bytes and padded as described by the SHA256 specifications;
- int be the set of intervals in which the $y_1, \ldots, y_m$ are modified in X;
- SHARound be the circuit that given a chunk of X and $h_0, \ldots, h_7$, the output of the previous round, produces new values $h'_0, \ldots, h'_7$ as described by the SHA256 specifications.

We say that $X = X_1, \ldots, X_n$ meaning that X is composed by n chunks of 64 bytes. The same holds for Y. Moreover, for simplicity, we define a function f that given Y, int, and $y_1, \ldots, y_m$ is able to reconstruct the original X. The statement ChunkPreImage that our implementation proves for each modified block is the following: $\exists y_1, \ldots, y_m$ s.t. $\mathsf{SHARound}(h_0, \ldots, h_7, f(Y_i, int, y_1, \ldots, y_m)) = h'_0, \ldots, h'_7$, where Y_i is the modified chunk, $y_1, \ldots, y_m$ is the witness owned by the prover and $h_0, \ldots, h_7, Y_i, int, h'_0, \ldots, h'_7$ are all public values. We remark that the verifier can compute the output of all SHA256 rounds on unmodified blocks and check that the final hash is equal to the value stored in the Merkle tree of the Bitcoin blockchain.

We explain the content of our tool describing how it works for a modified chunk X_i of X.

- `hash.cpp`. This source file implements the statement ChunkPreImage for use in Isekai. The main routine is:
  `void outsource(struct Input *input, struct NzikInput *nzik, struct Output *output)`
  that specifies the public input `input` of type `struct Input`, corresponding to the variables

[2] There are variant of these systems for specific classes of statement that have logarithmic verifiers. However, we cannot employ them for our application.

$X_i, h_0, \ldots, h_7$ and the secret input `nzik` of type `struct NzikInput` corresponding to the variables $y_1, \ldots, y_n$. `output` of type `struct Output` corresponds to the variables $h'_0, \ldots, h'_7$. The routine `outsource` will use the public and secret inputs to compute the intermediate hash of SHA256 on the current chunk and store it in `output`.

- `hash.h`. This header file specifies the types of the structures `struct Input`, `struct NzikInput` and `struct Output`. The structure `struct Input` has the following format:

```
struct Input {
 unsigned char tr[64];
 unsigned int h0[2];
 unsigned int h1[2];
 unsigned int h2[2];
 unsigned int h3[2];
 unsigned int h4[2];
 unsigned int h5[2];
 unsigned int h6[2];
 unsigned int h7[2];
 unsigned int start[64];
 unsigned int end[64];
};
```

and `NzikInput` the following: `struct NzikInput{`

```
 unsigned char deleted_data[DELETED_DATA_LENGTH];
};
```

The field `tr` contains the 64 bytes of Y_i, `deleted_data` contains the data $y_1, \ldots, y_m$, and the `h` values contain the output of the previous round of SHA256[3]. Isekai and Aurora work in the circuit model, so we have to fix an upper-bound to the maximum number of bytes that can be removed by a single 64 bytes chunk, that in our implementation is represented by the constant `DELETED_DATA_LENGTH` in the file `hash.h`. The arrays `start` and `end` represents the starting points and the end points of each interval in which the data are removed.

From Y_i, the routine `outsource` will first perform the string replacement using `deleted_data`, `start`, and `end` obtaining back X_i. X_i together with `h0`, ..., `h7` will be passed to SHARound to obtain $h'_0, \ldots, h'_7$ that will be put in `struct Output` that is: `struct Output {`

```
unsigned int h_out[8];
};
```

If the number of deletion intervals is less than `DELETED_DATA_LENGTH`, the remaining elements of the arrays `start` and `end` can be set to 0.

- `proofdel.py`. This source file contains routine to generate the inputs to the statement from a `transaction` file and `original_tx` file , to generate the circuits for Isekai, to create and to verify the proofs.

Analysis. To assess the performance of our code, we executed the following tests. A first test was done with a simple transaction with 64 bytes, in which only 4 bytes contained in the first chunk were deleted. A second test was performed on the coinbase transaction of the genesis block, deleting the 69 bytes of the Chancellor sentence. The third test was performed on the Bitcoin transaction indexed as "$db27236623f19ceaf8535407e74b5dfad613aef7d5558631f4837fd0f6d83c83$", that we call db2723tr in

[3] If the modified chunk is the first chunk we note that the `h` values are known and defined by the SHA256 specifications.

	Size (B.)	Modified chunks (num.)	Deleted data size (B.)	Prover (sec.)	Verifier (sec.)
Simple tr.	64	1	4	36.8	3.0
Chancellor tr.	204	2	69	82.9	6.1
db2723tr	283	3	76	123.9	9.2
Example tr.	3888	23	576	921.0	69.2

Figure 5.3: Performance of deletion using our Bitcoin sanitizer.

Table 5.3. The last test was performed on an ad-hoc `OP_RETURN` transaction with 3888 bytes, in which there are 16 `OP_RETURN` codes, in which we have deleted the data contained in the 16 `OP_RETURN`s.

The performance analysis reports the transaction dimension in bytes, the number of modified chunks, the number of bytes deleted by the entire transaction and the execution time in seconds of the prover and the verifier. The results are shown in Table 5.3.

Chapter 6

Contact Tracing and Blockchains as Shared Memory

Following recommendations of epidemiologists, governments are proposing the use of smartphone applications to allow automatic contact tracing of citizens. Such systems can be an effective way to defeat the spread of the SARS-CoV-2 virus since they allow to gain time in identifying potentially new infected persons that should therefore be in quarantine. This raises the natural question of whether this form of automatic contact tracing can be a subtle weapon for governments to violate privacy inside new and more sophisticated mass surveillance programs.

In order to preserve privacy and at the same time to contribute to the containment of the pandemic, several research partnerships are proposing privacy-preserving contact tracing systems where pseudonyms are updated periodically to avoid linkability attacks. A core component of such systems is Bluetooth low energy (BLE, for short) a technology that allows two smartphones to detect that they are in close proximity. Among such systems there are some proposals like DP-3T, MIT-PACT, UW-PACT and the Apple&Google exposure notification system that through a decentralized approach claim to guarantee better privacy properties compared to other centralized approaches (e.g., PEPP-PT-NTK, PEPP-PT-ROBERT). On the other hand, advocates of centralized approaches claim that centralization gives to epidemiologists more useful data, therefore allowing to take more effective actions to defeat the virus.

Motivated by Snowden's revelations about previous attempts of governments to realize mass surveillance programs, in this chapter we first analyze mass surveillance attacks that leverage weaknesses of automatic contact tracing systems. We focus in particular on the DP-3T system (still our analysis is significant also for MIT-PACT and Apple&Google systems).

Based on recent literature and new findings, we discuss how a government can exploit the use of the DP-3T system to successfully mount privacy attacks as part of a mass surveillance program. Interestingly, we show that privacy issues in the DP-3T system are not inherent in BLE-based contact tracing systems. Indeed, we propose a system named Pronto-C2 that, in our view, enjoys a much better resilience with respect to mass surveillance attacks still relying on BLE. The system is based on a paradigm shift: instead of asking smartphones to send keys to the Big Brother (this corresponds to the approach of the DP-3T system), we construct a decentralized BLE-based automatic contact tracing system where smartphones anonymously and confidentially talk to each other in the presence of the Big Brother. Pronto-C2 relies on Diffie-Hellman key exchange providing better privacy but also requiring a bulletin board to translate a BLE beacon identifier into a group element.

In particular, the Pronto-C2 system can be implemented using blockchain as a *shared memory* offering complete transparency and resilience through full decentralization, therefore being more appealing for citizens.

Part of this work will appear in the proceedings of the CoronaDef workshop of NDSS 2021.

6.1 Introduction

In 2013 Edward Snowden disclosed global surveillance programs [CHRT20] opening a worldwide discussion about the tradeoff between individual privacy and collective security. A common opinion of scientists after those facts is that the task of establishing standards to be used for cryptographic protocols should not be assigned to an organization that decides on its own, without providing the full transparency that such processes deserve.

SARS-CoV-2. A major threat is currently affecting humanity: the COVID-19 pandemic. The aggressiveness and fast spread of the SARS-CoV-2 virus have a strong impact on public opinion. Several governments are taking the most restrictive measures of the last decades in order to contain the loss of human lives and to preserve their economies. Fear is spreading, citizens are forced to stay home, many jobs have been lost, and more dramatically the number of deaths goes up very fast day by day.

Contact tracing. According to epidemiologists, a major problem with COVID-19 is that the virus spreads very quickly while current procedures to detect infected people and to find and inform potentially infected people are slow. When a new infected person is detected, too much time is spent to inform her recent contacts and to take proper restrictive actions. Commonly when a new infected person is discovered, by the time her recent contacts are informed they have had already a significant chance to infect others.

In order to improve current systems, many researchers are proposing automatic systems for contact tracing. Such systems can dramatically increase chances that recent contacts of an infected person are informed before infecting others. Essentially, whenever a person is diagnosed as infected, all her recent contacts (i.e., persons that have been in close proximity to the infected one) are informed immediately. This allows to promptly take appropriate countermeasures.

Automatic contact tracing (ACT, for short) is therefore considered an important component that in synergy with physical distancing and other already existing practices can contribute to defeating the SARS-CoV-2 virus.

Privacy threats. There are serious risks that ACT systems might heavily affect privacy. Citizens could be permanently traced and arguments like "If you have nothing to hide, you have nothing to fear" (Joseph Goebbels - Reich Minister of Propaganda of Nazi Germany from 1933 to 1945) are already circulating in social networks. Governments could leverage the world-wide fear to establish automatic contact tracing systems in order to realize mass surveillance programs. Motivated by such risks, several researchers and institutions are advertising to citizens the possibility of realizing automatic contact tracing systems that also preserve privacy to some extent. Such systems crucially rely on Bluetooth low energy (BLE, for short).

The BLE-based approach. BLE is a technology that allows smartphones physically close to each other to exchange identifiers requiring an extremely low battery consumption. Such communication mechanism avoids GPS technology and third-party devices like Wi-Fi routers or base stations of cellular networks. It is therefore a viable technology to allow the design of privacy-preserving ACT systems.

BLE-based tracing is used by Apple in a privacy-preserving system to find lost devices [Gre19]. Matthew Green in a interesting webinar with Yehuda Lindell [GL20] explicitly proposed to start with Apple's tracing system when trying to design a privacy-preserving proximity ACT system for citizens. Apple and Google have recently announced a partnership to provide an application program interface for exposure notification (GAEN, for short) [App20b] that can be used to include such features in smartphone applications.

In parallel with the Apple&Google initiative of GAEN, other BLE-based approaches very similar in spirit were proposed. Such BLE-based systems commonly rely on the use of pseudonyms that smartphones announce through BLE identifier beacons. After a short period of time, each smartphone replaces the already announced pseudonym with a (seemingly independent) new one. Each smartphone receives pseudonyms sent by others and stores them locally. Therefore, a smartphone will have a database of the announced pseudonyms and a database of the received pseudonyms. The central idea is that whenever a person is diagnosed with COVID-19, smartphones that have been physically close to the smartphone of the infected person should be notified and should compute a local risk scoring. In order to realize this, the smartphone of the infected person should use the above two databases to somehow reach out the smartphones that have recently been physically close to it. This communication is achieved through a backend server as follows. First the smartphone of the infected person will use the above two databases to communicate data to the backend server. The server could run some computations on data received from smartphones of infected citizens. The server will also use collected/computed data to answer pull requests of smartphones that desire to check if there is any notification for them.

Intuitively, the above approach through the unlinkability of the pseudonyms guarantees some degree of privacy. Despite the privacy-preserving nature of the BLE-based approach, the risk that such systems can be misused to realize mass surveillance programs remains a major concern that might slowdown the actual adoption of such systems. Indeed, most governments are not imposing the use of ACT systems.

Centralized vs Decentralized BLE-Based ACT. An important point of the design of a BLE-based ACT system is the generation of pseudonyms used by smartphones. Two major approaches have been proposed so far.

In a centralized approach pseudonyms are generated by the server. Each smartphone, during the setup of the ACT smartphone application, connects to the server and receives its pseudonyms. Therefore the server knows all the pseudonyms honestly used in the system. This is pretty obviously a clear open door to mass surveillance. Such dangers are discussed in [DP-20]. Currently the centralized approach is part of the protocols named NTK and ROBERT that are developed inside the Pan-European Privacy-Preserving Proximity Tracing (PEPP-PT) initiative [PEP20].

The decentralized approach breaks the obvious linkability of pseudonyms belonging to the same smartphone by letting the smartphone itself generate such pseudonyms.

While the decentralized approach has a better potential to protect privacy, the centralized approach has a better potential to provide useful data to epidemiologists.

Straight-forward decentralized BLE-Based ACT. The most trivial way to realize a decentralized BLE-Based ACT system consists of giving to the server the role of proxy that forwards to non-infected persons the pseudonyms of those infected persons that decide to upload their pseudonyms[1] after being detected infected. Therefore, everyone, including the server, clearly learns directly pseudonyms that have been used during the previous days by recently infected persons. Instead the pseudonyms generated by smartphones belonging to non-infected persons are not uploaded to the server and thus they are visible only to whoever was physically close to those smartphones. In terms of privacy, such straight-forward decentralized systems seemingly have a potential to offer a better protection compared to known systems that use the centralized approach. There are a few proposals based on the straight-forward decentralized approach, most notably Decentralized Privacy-Preserving Proximity Tracing (DP-3T, for short) and Private Automated Contact Tracing (MIT-PACT, for short).

Is privacy-preserving ACT a fig leaf? The unlinkability of pseudonyms advertised in BLE identifier beacons is completely useless if the BLE MAC address associated to a smartphone does not change in a synchronized way with the pseudonyms [BLS19]. Notice that iOS and Android make up (almost

[1]The actual information uploaded is a seed that generates the pseudonyms.

completely) the vast majority of the currently deployed operating systems for smartphones and have some serious restrictions on updating a BLE MAC address. In contrast, the smartphone application should obviously work in the background and should have control over the BLE MAC address so that this value can rotate along with the pseudonyms announced in the BLE identifier beacons. Therefore, because of such limitations of the vendors, it is absolutely problematic to realize BLE-based privacy-preserving smartphone applications that can practically (in the sense of usability, battery consumption, and so on) work on (almost) all currently used BLE smartphones, unless some flexibility is allowed by Apple&Google through updates of iOS and Android.

The move of Apple&Google. Interestingly, Apple&Google have released updates of iOS and Android providing GAEN to some "chosen" smartphone apps [2] resolving along with it also the MAC address linkability problem. However, the two features are seemingly connected, more precisely: if you want to implement a usable smartphone application (i.e., an application that runs in the background without battery drain on a very large percentage of the currently available smartphones) that needs to rotate the BLE MAC address synchronously with the content of the BLE identifier beacon then you must use their API and therefore you must use their approach for pseudonym generation and exposition.

This lack of flexibility generates some interesting consequences. First of all, the centralized approach does not seem to be implementable since it relies on pseudonyms generated by the server and then advertised in the BLE identifier beacon by the smartphone. Instead, the generation of pseudonyms can only happen inside the smartphone when using GAEN. Such mismatch implies that the decision of Apple&Google makes hard to realize the centralized approach to privacy-preserving ACT. Indeed, it is not surprising that some governments that originally had a bias towards centralization at some point decided to switch to the GAEN approach. Very sadly, GAEN also excludes better approaches that avoid replay attacks [Pie20a]. Snowden's revelations included memos confirming the existence of backdoors (e.g., see Dual_EC_DRBG) in standardized cryptographic algorithms [Wik]. It is therefore important to make sure that Apple&Google will not abuse such systems, and will not help governments interested in mass surveillance.

6.1.1 Our Contribution

Starting with the inspiring list of attacks presented by Vaudenay [Vau20a], in this chapter we first analyze the degree of privacy protection achieved by the DP-3T systems. In some of the attacks a government through its natural power controls (even partially) the server, the laboratories that detect infections and the national territory to realize mass surveillance programs.

We consider quite dangerous the fact that in the DP-3T systems (and all analogue systems) one can be traced even when walking alone, silently. Indeed, a passive antenna can detect the pseudonym without transmitting anything, and can later on check if the sniffed pseudonym belongs to the list of infected persons. It is easy to link the real identity of an infected person with the pseudonyms she used in the last two weeks. Indeed, such antennas can also be installed nearby any place where the citizen can be identified (e.g., showing an ID card or paying with a credit card) and this allows to connect pseudonyms to identities. We believe that this is an open door to help mass surveillance programs. Also other BLE devices that are in general used for other purposes (e.g., information kiosks) can be used to trace people. Obviously one can not expect that nothing else will be done with BLE except contact tracing, and thus preserving privacy while other uses of BLE continue is a necessary goal. Notice also that the use of active kiosks running precisely the BLE-based contact tracing protocol is actually recommended in [PAC20] (see Remark 1 in Section 6.6). Instead, we believe that they can be a source of privacy attacks. The lack of privacy with respect to such adversaries is a major vulnerability in the

[2] They provide only the part concerning the generation, rotation, and exposure of pseudonyms along with a flag to activate/dis-activate this service in the settings. There is no user application and neither a server collecting pseudonyms.

side of DP-3T and other analogue systems[3]. We stress that the issues exist regardless of the update of the MAC address of the BLE device. Technically speaking, the key weakness of the DP-3T system is actually a weakness of the straight-forward decentralized approach: asking smartphone applications to hand over the used keys/pseudonyms to the server is like asking citizens to kneel down in front of the Big Brother[4].

Next we present Pronto-C2, a new decentralized privacy-preserving automatic proximity contact tracing systems based on BLE. We show that our system is arguably more resilient than the DP-3T systems against mass surveillance attacks, while remaining useful for epidemiologists. Our system needs crucially a bulletin board, and this of course can be implemented through government servers if citizens trust the government. The bulletin board can be completely decentralized relying on blockchain technology. We believe that full decentralization can play an important role to help the work of epidemiologists since citizens obviously prefer to use their smartphones in ACT systems that are transparent and resilient to attacks, in addition to being privacy preserving.

6.1.2 High-Level Overview of Pronto-C2

Our main idea can be seen as a paradigm shift compared to the straight-forward decentralized approach. Indeed, instead of asking infected people to hand over their keys to the Big Brother as in DP3-T systems, we allow citizens to anonymously and confidentially call each other in the presence of the Big Brother. The way we do it is explained below.

In the 70s Merkle, Diffie and Hellman invented public-key cryptography. Starting with Merkle's puzzles, Diffie and Hellman proposed a key exchange protocol [DH76] (i.e., the Diffie-Hellman protocol) where two parties can establish a secret key K by just sending one message each on a public channel. A message consists of a group element in a setting where the so called Decision Diffie-Hellman assumption holds.

In our view, the most natural way to realize a privacy-preserving ACT system consists of having as pseudonym a group element that corresponds to a message in the DH protocol. This natural idea was also proposed to the DP-3T team by the github user a8x9 [a8x]. In order to actually realize such form of ACT system, one needs to solve the following two main problems.

Anonymous call: realizing a mechanism that allows an infected party to use K in order to call the other party in a secure and privacy-preserving way.

Shortening pseudonyms: making sure that the size of a group element fits the number of available bits in a BLE identifier beacon.

Calling (anonymously) the infected person. We solve the first problem by asking the infected party, after having received a proper authorization from the laboratory that detected the infection, to upload K along with the authorization to a bulletin board. The bulletin board can be just managed by a server as in the DP-3T systems, but it can be preferable to rely on decentralization through the use of blockchain technology, making the entire process transparent and reliable.

When implementing the bulletin board with a blockchain [5] then the authorization must be performed by a smart contact and thus the check should be accomplished uniquely with public information. For this reason, we suggest the use of digital signatures for implementing the authorization mechanism to upload data. In order to make the upload of K unlinkable with the real identity of the infected person, we suggest the use of blind signatures [Cha83]. The basic idea is that laboratories receive from the

[3] We will instead show that our Pronto-C2 system does not suffer from such drawbacks.

[4] We refer to the George Orwell book "1984", in which the Big Brother was the leader of Oceania and every citizen of Oceania was under constant surveillance by the authorities.

[5] In this chapter, when referring generically to a blockchain we always mean a permissioned blockchain (e.g., Hyperledger Fabric [ABB+18b]).

government some unpredictable activation codes that are then one by one given to infected persons. Then, an infected person connects to a service in order to exchange the authorization code with some blind signatures that will be useful to then upload on the bulletin board data associated to calls. In case of use of a blockchain to implement the bulletin board, this exchange of an authorization code with a blind signature is performed off-chain since the server will use a signature secret key and thus it can not be directly implemented by a smart contract.

Notice that the approach of Pronto-C2 is therefore completely different from the one adopted in the DP-3T systems. Indeed, while in the DP-3T systems the pseudonyms of the infected person are broadcast to everyone (or added to a Cuckoo filter by the server that then transmits the filter) we instead ask the infected party to send a message that is understandable uniquely by the party with which she was in close proximity (i.e., a message that can be computed in an unique way based on information in its possession). Therefore, K is more like a phone call where the infected party sends to the answering party the following message:[6] "Hello, it is you that were next to me... and I've just discovered that I'm infected".

Every person that is not infected will connect to the server (or to the blockchain) and will download the recently uploaded keys to search for K (data don't need to be stored, the search can happen while downloading data). Notice that there is a different key K to check for every BLE identifier beacon received in the last two weeks that has not been already discovered. This step should be preferably performed while the phone is connected to the charger and to a Wi-Fi network. Moreover, for those cases where the daily amount of data to download is excessive, one can think of specifying target states/regions in the country, in order to manage a restricted amount of information. In this case a call would also specify a corresponding state/region.

In addition to K, the infected person can also upload the root of a Merkle tree where the leaves include committed information (e.g., about BLE signal strength, location, body temperature) that later on the infected person might like to share with epidemiologists. The binding of the commitment is important to avoid that such data are adaptively changed. The hiding through a Merkle tree is important to leave the ownership of this information to the person until she decides to selectively disclose it.

We remark that avoiding that two smartphones with pseudonyms A and B upload the same K (this would leak some –most likely irrelevant – information), is straightforward: A could just upload $H(K||A||B)$ while B could just upload $H(K||B||A)$, where H is a cryptographic hash function.

6.1.3 Blockchain as Shared Memory

Current standards suggest at least 256 bits for a group element to safely run the DH protocol over elliptic curves. This size, however, exceeds the space available in a BLE identifier beacon. Moreover, we really stand for defeating mass surveillance attacks, and our system works with only a small overhead using 384 or even 512 bits. One might think to resolve the issue of the small space in a BLE identifier beacon by just resorting to very short (and therefore in our view too risky in case of mass surveillance attacks) keys [a8x20a] or by splitting the information into multiple identifier beacons that rotate quickly. Obviously Pronto-C2 can work smoothly with such workarounds, but since they all bring some issues, we propose a different approach that allows to use many bits for the group element while still remaining with one standard identifier beacon only.

In Pronto-C2, we decouple the group element from the pseudonym precisely like in operating systems a large amount of data is represented by a pointer. Recall that following also previous work, a value announced in a BLE identifier beacon should last only for a few minutes, to then be replaced by a new one. The smartphone will periodically generate new independent group elements for DH and will keep them locally. Since such group elements are too large to be sent in BLE identifier beacons,

[6] The Italian word "Pronto" stands for "Hello" and C2 pronounced in English stays for "it is you" in Neapolitan language as in the title of a very popular song of Nino D'Angelo [D'A83] (see also the movie [Lau83], min. 59:00).

the smartphone will upload them to a bulletin board. Again, our design is flexible and the bulletin board can be maintained by a server or alternatively be implemented with a blockchain. As above, we support the second option since it gives full decentralization and makes more citizens willing to participate, having more chances to defeat the virus. Notice that this generation of group elements is done only once in a while, and therefore can typically be performed when the smartphone is on charge and is connected to a Wi-Fi network.

In Pronto-C2 we decouple the group element from the pseudonym by setting the 128 bit[7] pseudonym as the address on the bulletin board of the corresponding group element. In other words, a pseudonym is a pointer to a *public shared memory*, therefore one can just refer to a short string to refer to an arbitrarily large amount of data [8]. Note that there is no downgrade of security from 256 to 128 bits since the pseudonym is used just as a pointer. By using as pseudonym a short representation of the group element, we need a different mechanism to implement the key exchange. Recall that the infected person must compute the key K and push it to the server, while the non-infected person needs to compute the key K to then check if it exists on the server. Starting from a short pseudonym every player will recover the actual group element from the bulletin board that records all group elements. This is a fast operation since the pseudonym is the address of the group element and thus there is no need to download a large amount of data or to do any expensive search.

6.1.4 Tracing

There are two quite different ways to trace.

Silent tracing. Pronto-C2 is clearly secure with respect to silent tracing. The key point is that Pronto-C2 is based on virtual anonymous calls originated from a recently detected infected person, and addressed to whoever has been in close proximity to her. Indeed, when a person walks alone and passes by a silent tracing device, the sole transmission of the pseudonym used in that moment by the smartphone does not allow to understand if later on that person is infected. There will be no key K corresponding to a key agreement in the silent tracing device that can be found in the list of virtual anonymous calls.

Shameless tracing. A government can also try to trace citizens by having on its territory many devices that behave as smartphones, therefore announcing pseudonyms with the hope of receiving a call or making calls in order to infer some information on the locations and identities of the citizens. It goes without saying that such attack is easier to detect compared to silent tracing. Indeed, the smartphone application could easily inform the owner at any time on the number of BLE identifier beacons that are currently received. Therefore, there is more room for citizens to realize the existence of malicious devices and ask police to destroy them and to identify the criminals that were trying to abuse the ACT system. Any government that would like to save its reputation convincing citizens to still use the smartphone application should take severe actions against such criminals. Obviously, if there is no prompt reaction of the government then citizens will feel that some attempts of mass surveillance are in progress and will simply switch off the smartphone application.

Notice that the only dangerous BLE devices are the ones that announce the very specific identifier beacon for the contact tracing system. There are specific codes to differentiate identifier beacons for different systems. Therefore, in our system, it is still completely fine (i.e., they do not have to be destroyed) to have on the territory devices (e.g., information kiosks) that use BLE to provide other services.

[7] This is the size for a pseudonym that is commonly allowed by BLE identifier beacons.

[8] A similar idea is used in IPFS [PL20].

Pronto-C2 is secure also against shameless tracing. Notice that with shameless tracing the infected user will upload a call for the active tracing device that was in close proximity. However, there will be no way to link calls coming from the same infected citizens when sending different pseudonyms as BLE identifier beacons. Therefore, unless we are in the extreme case where there is only one new infected person in a large area and in a significant amount of time, Pronto-C2 protects infected citizens from attempts to trace their movements through active BLE beacons.

Unlinkability over TCP/IP, timing, and other side-channel attacks. As in all ACT systems, users could be de-anonymized through the IP address when connecting to servers. Moreover, in Pronto-C2 when uploading a batch of group elements some attention should be paid so that they are not linkable. We therefore suggest the use of artificial delays and uploads of bogus data (i.e., dummy traffic) with the only purpose to confuse the adversary making harder any profiling attempt. We also discuss a simple solution to mitigate the above linkability issues using mixers. We assume that each user can select her own favorite mixer among several options that can belong to heterogeneous entities (e.g., political parties, large organizations defending civil rights). By doing so, users could pick their favorite options to protect their IP addresses when uploading their pseudonyms and their anonymous calls to the bulletin board and when downloading pseudonyms corresponding to the received BLE beacon identifiers. Moreover the user can send a batch of pseudonyms and calls since they will be mixed by the mixer that will also apply some artificial delays, and dummy traffic therefore guaranteeing some degree unlinkability (i.e., mixers that serve large communities give a satisfying degree of privacy, unlike mixers that serve only very few citizens). We give a more detailed description of this idea in Section 6.8. We remark that the server can also perform mixing and the privacy will be based on at least one among server and mixer behaving honestly. We stress that ACT systems currently deployed are affected by such issues and mostly ignore them, but still we prefer to discuss possible workarounds, even though they obviously introduce extra overhead.

Replacing DH with other key-exchange protocols. We have proposed the DH protocol because it is computationally efficient and has very low space requirements. Nevertheless, our design is flexible and one can use other key-exchange systems as long as there is just one message per party that is moreover independently computed from the other message.

Countermeasures to DoS attacks. Typical DoS attacks can be mitigated with pretty standard approaches, just to mention some: CAPTCHAs, proofs of work, anonymous tokens. We will discuss how to mitigate DoS attacks to the bulletin boards, by allowing regular (i.e., non-infected) citizens to upload a limited amount of pseudonyms, while infected citizens will be allowed to upload a very large amount of calls.

Removing old data from the bulletin boards (even from the blockchains). The entire information available on the bulletin boards does not disclose identities. Moreover, it does not allow any player that is not a sender nor a receiver of the call to link calls with pseudonyms. Nevertheless, in order not to overload servers with old information (e.g., anything uploaded more than 20 days ago), past data can be removed from the bulletin board pretty easily. If the bulletin board is managed by a server, then old data can just be deleted. If instead the bulletin board is realized through a blockchain, then we suggest that periodically the pointer to the genesis block moves forward to the next block. Essentially, the blockchain will always consist of the blocks generated in the last relevant time period (e.g., 20 days). Moreover, this process can be made even more transparent by uploading every 10 minutes on the Bitcoin blockchain the cryptographic hash of the blocks generated in the last 10 minutes. This allows everyone to constantly verify that the bulletin board is correctly decentralized and redacted.

Refuting a claim of the DP-3T team. Vaudenay in [Vau20a] showed a privacy attack to DP-3T proposing an antenna that can be used to eavesdrop the identifier beacons sent by smartphones. Some of our attacks follow the same idea even though we present them with a different syntax and focusing on a government that eavesdrops BLE communication as part of mass surveillance programs. The main point of this remark is that the DP-3T team answered to [Vau20a] in [DT20c] claiming that "This is a known attack vector inherent to all contact tracing systems, whether centralized or decentralized (SRE, Inherent Risk 1)". We refute this claim of the DP-3T team and indeed we contradict it by showing that Pronto-C2 is not affected by such attacks. It might be that the DP-3T team was implicitly referring to systems that follow the straight-forward approach only. This would imply that Apple&Google through GAEN are providing an ACT system that inherently suffers of privacy and security issues, and that could be exploited to help mass surveillance.

6.2 Related Work

In this chapter, we mainly focus on the security of the systems proposed by the DP-3T [DP-20] team. However, the attacks we present are significant to many other decentralized ACT systems such as MIT-PACT [PAC20], UW-PACT [CFG+20b] and TCN [TCN20]. Such decentralized ACT systems are very prone to be abused. Attacks can be carried out not only by the government, but also by unknown adversaries. These vulnerabilities have been acknowledged in [CFG+20b] (Section 3.1.3) where it is affirmed: "This can be abused for surveillance purposes, but arguably, surveillance itself could be achieved by other methods". As previously discussed, MIT-PACT [PAC20] even makes an explicit recommendation that active BLE devices should be placed on the territory by the government itself. Their intended purpose is to relay previously detected pseudonyms in order to warn users about possibly contaminated surfaces. However, they could easily be exploited for mass surveillance, being a perfect front for shameless tracing.

Several vulnerabilities of the DP-3T systems have been previously analyzed in various works [Tan20a, Pie20a, Vau20a, Vau20b]. Vaudenay [Vau20a, Vau20b] presents a detailed list of attacks against the DP-3T systems; some of the attacks in this chapter are indeed inspired to the ones of Vaudenay, but show with more emphasis the possibility to exploit such attacks for mass surveillance. The DP-3T team reacted to Vaudenay's work by presenting a public response to his attacks [DT20c] that does not object on their applicability, and sometimes tries to convey the message that those attacks are inherent to any decentralized approach. In this chapter we show that this is not true.

As part of a joint partnership, Google and Apple [App20b] recently released an update to their mobile operating systems to introduce a new set of exposure notification APIs that is subject to all the attacks shown in this paper for the low-cost DP-3T system. The new APIs are not open source (e.g., in Android the APIs can be accessed via inter process communication to the Google play service that is a closed source application executed in background) and cannot even be tested: in order to use the APIs you need authorization by Google and Apple.

Pietrzak [Pie20a] proposes solutions and mitigations to replay and relay attacks against the DP-3T systems. Furthermore, Pietrzak identifies the issue that users in the DP-3T system can easily provide digital evidence of contacts with infected users. Tang [Tan20a] observes that the DP-3T systems may be subject to identification attacks, and presents a comprehensive survey on proximity tracing systems. Furthermore, a subset of our attacks is taken into account in [BAL20], where the authors propose new attacks inspired by ours and the other previous papers.

Pinkas and Ronen [PR20], building upon a design similar to the DP-3T systems, propose a system with an improved resilience to relay attacks, a better verification of risks and other useful features.

The aforementioned works all focus on decentralized ACT systems. In contrast, there are several centralized proximity tracing systems, in particular TraceTogether [Tra], adopted in Singapore and ROBERT [IPT20b], designed by Inria and Fraunhofer (a French and a German research institution

respectively).

In [Fra20] the authors review the most prominent European proximity tracing systems, DP-3T, NTK, and ROBERT, analyzing the different adversarial models assumed by each system.

WeTrace [CFG+20a] proposes a system [9] based on the use of public-key cryptography, similarly to Pronto-C2. Public keys are exchanged over the BLE channel and to solve the problem of having only 16 bytes available in the BLE beacon, users broadcast only the first 16 bytes, while the remaining bytes can be retrieved by querying a server. In order to access the full public keys efficiently, these data are indexed by hashing (a part of) the first 16 bytes obtained via BLE. (i.e., using the first 16 bytes as an address to find the remaining bytes of the key, similar to the pointers proposed in Pronto-C2). An infected user A who wishes to report, uploads messages encrypted with the public keys related to close contacts she had. These encrypted messages are independent of the public key A was broadcasting while being in contact with the recipients of the messages. While this might be beneficial for A's privacy, it can be severely affected by false positives attacks. Indeed a user B may be alerted consequently to an upload performed by an infected user TSen who did not come into contact at all with B, but just received B's key trough other means.

Another system similar to Pronto-C2 that appeared online on April 2020[10] is TraceCORONA [SSL20]. TraceCORONA is based on the exchange of exposure tokens that can be computed using the Diffie-Hellman protocol. Currently there are no papers associated with this ACT application proposal, therefore we do not have technical details to properly compare this solution with Pronto-C2.

After a preprint of this chapter appeared on ePrint, Inria published on Github a new ACT system named DESIRE [IPT20a], which is also based on the Diffie-Hellman key exchange scheme. After being tested positive, a user uploads data related to the encounters he had during the previous days. As in Pronto-C2, such data is computed hashing the shared key along with an information depending on which of the two users is creating the report. This guarantees that if two users A and B have been in close proximity, and both of them end up being positive to SARS-CoV-2, they will send two different values to the server, making it impossible for the server itself to infer that A and B have been co-located. Differently from Pronto-C2, the users' at-risk status is computed on the central server.

Interestingly, in [Sei20] Seiskari shows that the DP-3T low-cost system (that is even referred as "semi" decentralized) is attackable in practice. In particular he shows that positions in the prior two weeks of new infected users can be tracked by any third party (i.e., not only the government!) who can install a large fleet of BLE-sniffing devices. Notice that such devices can be completely passive (i.e., they do not broadcast an identifier beacon) and therefore are hard to detect. The attack is therefore hard to mitigate and inherent in the design of the DP-3T low-cost system. The github repository includes a Proof-of-Concept implementation of such BLE sniffer that succeeds against the low-cost DP-3T system.

Baumgärtner et al. [BDF+20] provide empirical evidence for the two main risks of the Apple&Google design, namely, as in DP-3T, tracing of infected users and replay/relay attacks.

6.3 Threat Model

In this section we present the adversary goals, capabilities, and the threats covered and not covered in this chapter. We remark that this chapter introduces several distinct attacks against automatic contact tracing systems and for each of the attacks the adversary may have different goals and capabilities and collude with different entities. So, the following discussion is general in nature.

Adversary goals and threats. An adversary attacking the privacy of the system wishes to *trace* users. By tracing users we mean that the adversary can link different locations visited by the same

[9] We became aware of WeTrace only in mid June 2020, informed by Adrienne Fichter.

[10] We have been informed of the existence of such system only in August 22nd 2020.

user. This is irrespective of the fact that the adversary may not know the real identity of the user who made such visits.

Notice that, due to the fact that pseudonyms in BLE packets do not change during a time slot, tracing in these periods is inevitable. Generally, we will not deem inevitable attacks as a threat. For instance, an adversary that can place a different smartphone at each location of the space and at each time window[11] can know where and at which time each diagnosed person has been in the case that there has been only a single diagnosed person; or can know at what time one of the smartphones under his control met a diagnosed person. These threats are inherent to any automatic contact tracing system.

An adversary attacking the privacy also wishes to *link* locations visited by users, both diagnosed and non-diagnosed, to their real identities. For instance, the adversary may attack the privacy by attempting to link information uploaded by the user to a server through the user's IP address.

An adversary attacking the integrity wishes to falsely alert users of having been in contact with a diagnosed person; replay and relay attacks fall in this category.

Adversary capabilities. We will consider different threats that involve adversaries of different capabilities. The adversary may place an arbitrary number of listening BLE devices at arbitrary locations. Such devices may operate exclusively in reception mode over the BLE channel. In this case, we say that the adversary is *passive* and is performing a silent tracing. An adversary may also try to trace citizens by placing many hidden devices that behave as regular smartphones. In this case, the adversary is *active* and is performing a shameless tracing. As explained in Section 6.1, shameless tracing can be easily detected by citizens that will react uninstalling the application.

An adversary may instead use smartphones belonging to actual citizens (e.g., policemen) to collect information about citizens without allowing them to detect such attack.

For some of our attacks we will consider an adversary that can even corrupt the server and/or the health authority.

Types of tracing attacks. Many of the proposed attacks (cfr., Sections 6.4.1, 6.4.2, 6.4.3) deal with tracing the movements of infected individuals over the contagion time window. Let us consider the strongest possible adversary who may try to trace infected users that is, as specified above, an adversary using active devices (i.e., behaving as regular smartphones) and colluding with the server. We now evaluate what is the highest level of privacy protection that can be guaranteed to infected users in this scenario. Note that an active adversary Adv is completely indistinguishable from a regular user of the system, this means that whenever Adv comes into contact with an infected user U who decides to upload data to the system, Adv will be alerted as prescribed by the system itself. In addition, Adv gets to see all the data uploaded by U to alert all the users she came into contact with. Suppose that Adv has placed a series of active devices over a certain territory, then for each of such locations where U has been over the contagion time window, U will upload data to server in order to alert Adv's devices. This means that Adv will certainly know which of his devices has been in proximity of an infected user and when.

Therefore, the best we can hope for in this scenario is that Adv cannot know whether data related to different locations are relative to the same individual. In this case, a certain degree of privacy is provided to infected users whose movements remain hidden within the set of movements of all other diagnosed people who uploaded data during the same day.

More specifically, we say that a protocol enjoys *partial protection* (with respect to passive or active adversaries) when a (passive or active) adversary who placed (passive or active) devices at two different locations X and Y cannot figure out whether X and Y have been visited by the same infected user. This also implies that if Adv received one alert related to position X at time t_1 and one related to position

[11] We mean that for each location and for each time window the adversary initializes a new smartphone with the automatic contact tracing application under his control.

Y at time $t_2 > t_1$, the adversary cannot know which user visited X at time t_1 and which user visited Y at time t_2. (Indeed, if Adv could know which user visited resp. X and Y could also know whether X and Y have been visited by the same user, in contradiction with the *partial protection* property.)

Of course, in general, there may be additional information that helps Adv to disambiguate. For instance, consider the scenario in which a pseudonym is listened at location X at time t_1 and another one is listened at location Y at time $t_2 > t_1$. If no other pseudonym has been listened at nearby locations at times $< t_1$, there are two possible explanations: 1) a user turned on his phone at location X at time t_1 and then moved to position Y at time t_2; 2) a user U_1 turned on his phone at location X at time t_1 and then, at time t_2, U_1 turned his phone off while another user U_2 turned his phone on. In this simple scenario, it seems obvious that the first case is more likely than the second one. However, disambiguating gets more difficult as the number of infected individuals and locations increases.

We say that an ACT system enjoys *full protection* from tracing attacks w.r.t. a certain passive adversary Adv, if Adv is not able at all to trace the movements of infected individuals during the contagion time window. To be more specific, even if there is a single infected user U, Adv does not get to know even one single location visited by U during the contagion time window. Notice instead that *partial protection* with respect to passive adversaries does not offer any guarantee in the case there is only a single infected user, in particular all the movements of such user could be leaked.

Furthermore, U may pass nearby such devices both when she is alone or when some other users of the system are also there. In the first case, U may not upload any data related to the period of time she was alone since there is no one to be alerted, while in the second case an alert should be sent to whom has been in contact with U. For this reason, an ACT may exhibit different levels of resilience to tracing attacks depending on the actual encounters the user had, in particular: it could protect infected users from being traced in any case (i.e., it provides full protection) or only for the periods of time they have been alone. We name the latter as *solitary protection* from tracing attacks. Obviously, *full protection* implies *solitary protection*.

Threats not addressed. We do not consider threats at the BLE layer such as using power signal and other side-channel information to identify users or issues at the operating system level.

6.4 Privacy Attacks for Mass Surveillance

Mass surveillance is an activity put in place to watch, even discontinuously, over a substantial fraction of the population by monitoring, for example, their movements and/or habits.

Even though decentralized solutions guarantee, in general, better privacy compared to centralized ones, mass surveillance is still a possible threat and must be mitigated as much as possible when introducing new intrusive technologies.

In the following paragraphs, we present several possible attacks towards contact tracing systems which, when successful, undermine users' privacy, eventually contributing to mass surveillance activities. Furthermore, we evaluate and compare the resilience of our Pronto-C2 system (see Section 6.7) and the DP-3T systems against such attacks.

Our attacks are inspired by the works of Vaudenay [Vau20a] and by the issues reported in the DP-3T's git repository [a8x20b, a8x20a]. We carefully take into account these issues and attacks to illustrate more precise scenarios unveiling significant mass surveillance attacks.

Both the DP-3T systems, and Pronto-C2 protect the privacy of non-diagnosed people and protect the privacy of diagnosed people out of the contagion time. Since these systems are the main focus of this chapter, in some of the following attacks we will assume that the adversary only attacks the privacy of diagnosed people during the contagion time window (e.g., roughly, the last two weeks before being tested positive). However, we remark that privacy protection is important for all users at any time, and

ACT systems might strongly violate privacy, especially when there is an extremely centralized design and when there is collusion among the various authorities of the system.

6.4.1 Paparazzi Attack: Tracing Infected Users with Trusted Server

This attack is similar to the Paparazzi attack reported in [Vau20a]. The main difference between the two, is that the one of [Vau20a] has the purpose of de-anonymizing infected users, while here we focus on building a mass surveillance infrastructure to trace citizens[12].

- **Attacker's capabilities**: The attacker Adv is anyone with enough economical resources. Adv has the ability to install, in a sufficiently large number of different locations, passive BLE devices. The only capability of a passive device is to operate over BLE channels in reception mode. We also assume that such devices are provided with enough memory to store a significant amount of received data (i.e., pseudonyms and auxiliary information).

- **Attack description**: The passive devices record the observed pseudonyms along with a fine-grained time log. The location of each device is fixed and determined by the attacker Adv. When a user B is tested positive and uploads data into the ACT system, the system itself provides related data to all users. Adv then combines these data with his logs. Furthermore, the attack is practically undetectable by the users since the BLE devices operate only in reception mode.

- **Attack's outcome**: Adv traces the infected users over the contagion time window. The attack is considered successful if the system fails to provide *full protection*. Variants of the attack can be considered, where Adv is interested in breaking *solitary protection* or *partial protection* respectively. We name the first variant as Solitary Paparazzi attack and the second one Partial Paparazzi attack.

6.4.2 Orwell Attack: Tracing Infected Users with Colluding Server

Orwell attack differs from Paparazzi attack only for the capabilities of the attacker.

- **Attacker's capabilities**: The attacker Adv is the same as in Paparazzi attack. However, in addition, Adv can collude with the server. Note that the server could be under a significant influence of the government.

- **Attack description**: Adv is analogous to the one described in Paparazzi attack. The only difference is that, along with data provided to all regular users, Adv receives all data that are in possession of the server.

- **Attack's outcome**: The outcome is analogous to the one of Paparazzi attack, we adopt also here the same naming convention for the variants of the attack.

6.4.3 Matrix Attack: Shameless Tracing of Infected Users with Colluding Server

- **Attacker's capabilities**: The attacker Adv is identical to the one of the Orwell attack in terms of the information he has access to. On the other hand, Adv's devices are active and can actively

[12]This kind of attack was described in [SSL20] when was discussed the possible attacks against GAEN.

send messages over the BLE channel.

- **Attack description**: Adv operates similarly to what has been shown in the previous attacks. Adv combines the data in his possession with the ability to actively send messages of the contact tracing protocol over the BLE channel in order to trace infected citizens over the contagion time window.

- **Attack's outcome**: Adv traces the infected users over the contagion time window. The attack is considered successful if the system fails to provide *partial protection*.

6.4.4 Brutus[13] Attack: Creation of Mappings Between Real Identities and Pseudonyms

- **Attacker's capabilities**: The attacker Adv consists of the server and the health authorities colluding together.

- **Attack description**: Adv exploits the authorization mechanism, also used to avoid uploads of false positives, to find a mapping between the real identity of a user B and her uploaded data.

- **Attack's outcome**: a mapping between the real identity of B and her uploaded data.

Every ACT system where the authorization mechanism grants a user B permission to upload data forwarding to the authentication server some data (e.g., an activation code) provided to B by the health authority, is vulnerable to this attack. Indeed, the health authority, who is aware of the real identity of B, can communicate the mapping between the activation code and the real identity of B to the server, which can in turn derive the mapping between this code and data uploaded by B. The authorization mechanism is not made explicit in many relevant proposals [PAC20, PR20]. A reason advocated for this choice is the flexibility to different deployment scenarios. However, we want to point out that the way this check is performed reflects into serious implications on users' privacy.

6.5 Other Attacks

6.5.1 Bombolo[14] Attack: Leakage of Contacts of Infected Users

- **Attacker's capabilities**: The attacker Adv consists of the server and the health authorities colluding together.

- **Attack description**: When users are tested positive, they upload data to the system. The attacker uses such data to extract information related to contacts among infected users and the number of contacts of an infected user.

- **Attack's outcome**: Adv succeeds in computing data as specified in the attack description.

[13] Marcus Junius Brutus was a close friend of Julius Caesar, who took a leading role in his assassination. His name has become synonymous with severe acts of betrayal.

[14] Franco Lechner, best known as Bombolo, was an Italian comedian. His characters usually played hilarious but harmless jokes.

Systems in which the infected users upload an encoding of the observed pseudonyms are more prone to this attack since the content and the amount of communicated data depend on the actual number of experienced contacts. One could think to mitigate this issue by putting a bound on the number of contacts that a user can notify. However, it is not evident what is the appropriate value for this bound to effectively fight the pandemic. Also, co-location of infected users is more likely to be exposed since infected users who met each other might end up reporting some linkable information. If at some point two infected users met each other, the information that these users sent to the server may enable the reconstruction of clusters of infected users who have been co-located. Nevertheless, such attacks are ineffective to link multiple locations visited by a given user and thus it is hard to imagine how such leakage could be exploited by mass surveillance attacks.

6.5.2 Gossip Attack: Proving Contact With an Infected User

This attack deals with the possibility to exploit ACT in order to produce plausible digital evidence of an encounter. An attack of this type against the DP-3T systems has already been reported by Pietrzak [Pie20a]. Starting from Pietrzak's work, we give a formulation of such attack against a general ACT.

- **Attacker's capabilities**: The attacker Adv has the same power as a regular user. Additionally, Adv might get access to a service making him able to prove the ownership of some data at a specific time (e.g., a blockchain).

- **Attack's outcome**: Adv provides a plausible evidence of having met an infected user B before B declared himself as positive through the ACT system.

Turning Gossip attack into a feature. Suppose that, due to the pandemic, laboratories are overwhelmed by requests for tests. In this scenario, having a way to prioritize the requests could be certainly useful. Indeed, there could be malicious users trying to fake risk notifications so that they eventually get tested, even if it is not actually needed.

To address this issue, one could leverage the Gossip attack as a feature. Laboratories could give a higher priority to users who are able to provide a plausible evidence of having met an infected individual. Depending on the system, a malicious user attempting to provide such fake proof would need the collaboration of someone who actually observed at least a pseudonym of an infected user. Such complications might reduce the noise of malicious users trying to create a fake plausible evidence. Therefore, prioritizing users with plausible (though not formally provable) evidence can be a concrete strategy for a health system.

This feature could be also very useful in assuring that reliable data are provided to epidemiologists. Such data are mainly related to encounters between infected individuals, therefore, providing evidence of these encounters could help to ensure that data provided to the epidemiologists are more reliable.

6.5.3 Matteotti[15] Attack: Putting Opponents in Quarantine

- **Attacker's capabilities**: The attacker Adv colludes with the server and the health authority. In addition, Adv can place passive BLE devices at selected locations.

[15] Giacomo Matteotti was an Italian socialist politician who openly denounced the electoral fraud committed by Fascists. He was kidnapped and killed by Fascists. The day he was murdered, Matteotti should have taken a speech at the parliament in which he would have disclosed significant scandals about the Duce.

- **Attack description**: The aim of Adv is to produce false alerts causing non-at-risk users to get tested.

- **Attack's outcome**: A non-at-risk user is erroneously alerted and declared as positive.

We motivate the attack with the following example. In the vast majority of world's country e-voting is not currently deployed, and, also at parliamentary level, voting is always held in presence. Suppose that a law, proposed by the government, risks not to get the approval of the parliament for very few votes. Then a malicious government could attempt to falsely report hostile parliamentarians as positive. Let B be a hostile parliamentarian. Hidden passive BLE devices could be put in place near the house of B during a given period. These BLE devices will intercept the pseudonyms $\mathsf{EphID}_{\mathsf{B}}$s of B and the pseudonyms $\mathsf{EphID}_{\mathsf{TSen}}$s of TSen, a person who lives with B. Then the government will add the $\mathsf{EphID}_{\mathsf{TSen}}$s to the list of users that will be notified as at-risk users. It is very likely that the next day TSen will go to get tested. At this point, if the test of TSen is positive, since there is a good chance that B and TSen will be in close proximity during the given period, the malicious health authority, colluding with the government, could issue an order of quarantine for B so that B will be unable to join the next parliament session.

We emphasize that in this attack, the adversary, that controls the server, has the power to generate false-positive notifications for a target user.

6.5.4 Replay Attack

- **Attacker's capabilities**: The attacker Adv is anyone who is capable of recording and broadcasting pseudonyms.

- **Attack description**: Vaudenay [Vau20a] and Pietrzak [Pie20a] discuss attacks in which Adv, who collects a pseudonym at location X where the probability to meet an infected person is higher, can then broadcast those pseudonyms to users at a different location Y. This attack is denoted as replay attack when the listened pseudonyms are broadcast at a later time slot and is the only case that we consider in this work[16].

- **Attack's outcome**: Users at location Y will be notified a risk even though they have been never in contact with infected people at location X.

6.6 Brief Description of DP-3T

In this section, we briefly overview the DP-3T systems as reported in the white paper [DP-20]. We describe two versions of the system: the first one, termed as "low-cost", is more efficient but provides lower privacy guarantees than the second one, which is termed "unlinkable". There is also another design proposed by the DP-3T team to provide better privacy requiring the users to secret share the ephemeral identifiers to be transmitted. Unfortunately, it seems that according to the DP-3T team this design is not practical. Indeed, the split of an identifier beacon among multiple different packets is analyzed by the DP-3T team in [DT20a] where it is remarked that there are several issues at the BLE layer: splitting a beacon identifier among multiple different packets increases the load on the battery as the CPU has to be woken up more frequently.

[16] If the attack is completely run in the same time slot, then any solution inherently requires some location information (e.g., by GPS) or problematic assumptions on time synchronization, therefore we do will stick with replay attacks only since they can be defeated without adding assumptions or penalizing privacy.

Low-cost design. As in every straightforward decentralized ACT system, smartphones broadcast locally generated ephemeral pseudonyms (EphIDs) via BLE advertisements.

Whenever a smartphone detects an incoming EphID, it locally stores this pseudonym EphID along with a coarse time information and every data which might be needed later to compute the risk of contagion (e.g., signal strength, duration of the contact). As the word ephemeral suggests, the pseudonyms are periodically changed to prevent tracing.

All the EphIDs that a device will ever generate can be deterministically derived from a short uniformly random secret key sk_0. At each day t, a new secret key is derived as $\mathsf{sk}_t = H(\mathsf{sk}_{t-1})$ where H is a cryptographic hash function.

Starting from sk_t the whole set of EphIDs for day t, is determined partitioning in 16-byte chunks a string whose length depends on how frequently the EphIDs are changed. Such string is computed as $\mathsf{PRG}(\mathsf{PRF}(\mathsf{sk}_t, c))$ where PRF is a pseudo-random function, c is a fixed public string, and PRG is a stream cipher. The EphIDs obtained with this procedure will be eventually broadcast in random order.

When a user is tested positive, she uploads the pair (sk_t, t) to a backend server which is trusted to provide this information to all other users and to check that the uploads are performed by authorized users, therefore preventing the dissemination of false positives. In [DT20d], three candidate authorization mechanisms are proposed. After this step, the infected user's device disappears from the application scenario and her device generates a completely new random secret key sk_0.

Each user can periodically query (e.g., at the end of the day) the backend server in order to get the new pairs that have been added to the system. Given these pairs, the device can generate the corresponding values EphIDs seeking for matches in its local contact database. If a match is found, the risk of infection is computed given the auxiliary information and the user is notified when needed.

Unlinkable design. In order to get better privacy guarantees at the cost of a larger volume of downloads and storage space needed by the smartphone, the DP-3T team also proposes a slightly different design which they term unlinkable.

In this design, the EphIDs are randomly and independently generated in the following manner: when a new ephemeral pseudonym is needed, the smartphone generates the ephemeral pseudonym EphID_i as $\mathsf{TRUNCATE}_{128}(H(\mathsf{seed}_i))$.

Smartphones store all the seeds used in a relevant time window (e.g., 14 days). When a patient is tested positive, she can selectively decide which pseudonyms she wants to communicate to the server (e.g., she can exclude pseudonyms used in the presence of a specific person).

After this decision has been made, the smartphone uploads the set composed by the selected pairs (seed_i, i). Upon receiving them, the server computes $H(\mathsf{TRUNCATE}_{128}(H(\mathsf{seed}_i))||i)$ for each pair and inserts it in a Cuckoo filter [17]. Such filters are generated and made available to the users on a regular basis.

Each smartphone uses these filters to determine if contacts with infected individuals occurred. In this regard, the smartphone checks the inclusion into the filters of all its recorded ephemeral pseudonyms.

6.6.1 Security Analysis of the DP-3T Systems

In this section we describe the security of both designs of the DP-3T team with respect to the attacks proposed in Section 6.4 and 6.5.

The low-cost design of DP-3T is vulnerable to Paparazzi attack. It is not difficult to imagine the feasibility of such an attack, as an example, one could consider a company with many stores spread over the territory. This corporation can have an interest in tracing infected costumers, even if it is not

[17] A Cuckoo filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not.

particularly interested in their health conditions, in order to use their movements to perform accurate profiling without costumers' consent.

What is needed is merely the capability to install, in a sufficiently large number of different locations, passive BLE devices recording the received EphIDs. The attack is carried out as follows. The attacker Adv controls a set of passive devices $\{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$.

1. Each passive device D_i collects the information of people that pass nearby D_i, the information stored consists of a set of pairs $(\mathsf{EphID}_j, \tau_j)$, where EphID_j is the pseudonym of a user that passes near D_i and τ_j is a fine-grained time log.

2. At the end of the day, Adv downloads the secret key of each infected user from the server and collects all data from each device D_i.

3. Adv checks if each collected EphID_j is generated starting by a secret key sk_j downloaded from the server.

4. Adv tracks the infected individuals who passed nearby the passive devices over a given contagion time window.

In the scenario we envision, the amount of gathered data can be considerably large, thus resulting in a possibly very fine-grained tracing.

The key issue of the low-cost design, leading to the applicability of Paparazzi attack, lies in the fact that when the secret key of an infected person is added to the system everyone can derive all the related EphIDs, enabling the linking of pseudonyms to infected individuals. We point out that this attack is practically undetectable, at least at the application level, since the devices do not need to propagate any signal. Given the huge impact that this easy-to-deploy attack can have on users' privacy, the DP-3T's low-cost design appears utterly unsuitable for practical deployment, unless one wants to give up on protecting citizens from mass surveillance attacks.

Unlinkable design of DP-3T is vulnerable to Orwell attack. Since the Cuckoo filter allows users to only test inclusion of seemingly uncorrelated EphIDs in the filter itself, the unlinkable design succeeds in preventing the Paparazzi attack. However, the claim that "infected people in the unlinkable design are not traceable", as affirmed in [DT20b] is oversimplified and requires a deeper treatment. In fact, such claim is true only with respect to attackers who do not cooperate with the server. Considering also the fact that governments might have control over the servers, an attack similar to the one described for the low-cost design can be put in place.

The devices listening on the BLE channels could be deployed or hidden in many ways. As an example consider smart kiosks, which are already used in many cities to provide useful functionalities to the citizens. For the purpose of the description, we will refer to all possible passive devices as kiosks. The attack works as follows:

1. Each kiosk D collects the information of people that pass near D, the information stored consists of $(\mathsf{EphID}_j, \tau_j)$ where the EphID_j are the pseudonyms of the users that pass near D and τ_j is a fine-grained time log.

2. Adv, that controls the kiosks and colludes with the server, obtains from the server all the seeds of the infected citizens.

3. Adv matches the EphIDs of records stored in the kiosks with the ones generated from the seeds of the infected individuals, thus tracing the infected individuals who passed nearby the kiosks over a given contagion time window.

The element of centralization in DP-3T requiring the server to compute the Cuckoo filter of the EphIDs, enables mass surveillance with low overhead. Moreover, it is almost impossible to determine if a process of surveillance is actually active or not.

Another important point is that governments can do a further step associating a pseudonym to the real identity of an infected user: whenever there is a police checkpoint to control people, the police can be instructed to collect EphIDs and associate them with the name and surname of the controlled persons. When a person is tested positive, the government can check data collected by the police. If one of the EphIDs comes from the seed of an infected person B, the governments can obtain all the movements of B during the contagion time window.

The same thing can happen when a citizen gets tested for SARS-CoV-2. In fact the tests are typically performed after some form of identification. If a citizen B goes to a laboratory and the smartphone application of B is active in the laboratory, EphIDs of B can be detected by a kiosk. If B is eventually tested positive and B uploads the seeds related to time in which he visited the lab, a match between B's real identity and his movements during the contagion time window can be easily exposed.

Remark 1. The idea of having kiosks spread over the territory could seem somewhat artificial. However, as stated by MIT-PACT [PAC20], it is possible to justify kiosks as a way to add functionalities to contact tracing systems. In particular, the authors of MIT-PACT state that there should be a way to inform persons if a surface can be contaminated due to the prior presence of an infected individual. Therefore, in their system, kiosks actively participatea in the protocol registering and relaying pseudonyms of people who have been in close proximity to the kiosks. By doing so, the kiosks could inform people about the risk of having been in contact with a contaminated surface. This system could provide an easy way to justify the deployment of kiosks in location where there is no clear need for them, thus facilitating the tracing of citizens disguising it as a service.

DP-3T systems are vulnerable to Matrix Attack. Since both designs of DP-3T provide no countermeasure to the linking of infected users' pseudonyms, it clearly follows that they are vulnerable to the Matrix attack. In particular, the additional capability of placing active devices over the territory is not even needed by an adversary succeeding in the attack against both designs.

DP-3T systems withstand Bombolo Attack. DP-3T and similar systems are not affected by this attack. Indeed, the data which are sent to the server are independent on the actual encounters the infected user had. Then, even if the attacker colludes with the server and the health authorities, the attacker cannot co-locate the seeds published on the server or infer the number of contacts infected users had.

DP-3T systems are vulnerable to Brutus attack. DP-3T proposes three candidate authorization mechanisms [DT20d]:

1. Simple Authorization Codes, in which the server generates authorization codes that are distributed to infected users after a positive test;

2. Activated Authentication Codes, in which the authentication codes are assigned at the testing time and only if the user is positive to SARS-CoV-2, the code is activated;

3. Data-Bound Authorization, in which the users commit at test time the data to upload to the server, if the user is positive to SARS-Cov-2, and then the health authority authorizes the upload of the data.

It is simple to notice that the three ways proposed by the DP-3T team are subject to the attack, and none of these mechanisms addresses the problem of collusion between the server and the health

authority. Indeed the data uploaded by an infected user can be always related to a single authorization code and then to the identity of the infected user.

DP-3T systems are vulnerable to Gossip Attack. An attack of this type against the DP-3T systems has already been reported by Pietrzak [Pie20a]. As plausible evidence of an encounter with a user B, A proves to have been in possession, at a time $t_1 < t_2$, of the pseudonym EphID_B of B, who, after having been tested, reported himself as positive to the ACT system at time t_2. The attack is really straightforward and it is instantiated as in [Pie20a]. Whenever A receives a pseudonym from a user B, he commits it to the Bitcoin blockchain. If B is later diagnosed infected and decides to upload his data to the system, A could then prove that he knew the pseudonym of B prior to this upload. To do so, A just needs to open the commitment on the blockchain. This procedure works in the same way for both designs of DP-3T, since the revealed EphID can be easily matched both with the published filters and secret keys. Notice that there is no guarantee about the fact that A himself received the pseudonym over the BLE channel. For example a device D in another (even remote) location could have committed the pseudonym and transferred its opening to A, by e-mail. However, in this case the attacker is actually the pair (A,D), who indeed met B. As noted in [Pie20a], the attack becomes a more serious threat if coupled with de-anonymization of B.

As we note in Section 6.5.2, it is possible to consider this attack as a feature, but it is very problematic in the DP-3T systems. The DP-3T white paper [DP-20] proposes that users, who are willing to do it, can share additional data with epidemiologists to help them in their analysis. Such additional data are mainly related to encounters between infected individuals, therefore, providing evidence of these encounters could help to ensure that data provided to the epidemiologists are more reliable.

Even though in the DP-3T systems it is possible to provide a plausible evidence of being at risk by leveraging the Gossip attack as a feature, it seems, at least at a first glance, that it would not be easily scalable to a considerable portion of the users.

The DP-3T team does not mention any explicit procedure to take advantage of this feature. However, the actual utility to provide additional data to the epidemiologists may be seriously compromised if the Gossip attack is not taken into account as a feature. The way the functionality to help epidemiologists is implemented, at least as in the current version of the DP-3T white paper [DP-20], presents some shortcomings. Indeed, users who want to give a further help in fighting SARS-CoV-2 anonymously communicate data related to contacts they had with infected users. However, in both designs, the system does not provide a mechanism to verify the legitimacy of the alleged contacts. Furthermore, there is also the need to trust the correctness of any additional metadata provided by users, although this seems an inherent problem.

The unlinkable design of DP-3T is vulnerable to Matteotti attack. Even though the unlinkable design solves in part the issue of linkability of the pseudonyms, the attacker Adv that controls the server gets more power, since Adv can add in the Cuckoo filter every EphID that Adv gets to know. This can cause additional false positives.

If Adv observes $\mathsf{EphID}_\mathsf{B}$ and $\mathsf{EphID}_{\mathsf{TSen}}$ in the same location and during the same time slot, then Adv adds $\mathsf{EphID}_\mathsf{B}$ and $\mathsf{EphID}_{\mathsf{TSen}}$ to the filter. The probability that, after checking the filter, both B and TSen are notified a risk is high since B will find $\mathsf{EphID}_{\mathsf{TSen}}$ in the filter as well as TSen will find $\mathsf{EphID}_\mathsf{B}$. Let us assume that B is the target of the attack. At this point, if B goes to a laboratory to get tested, the health authority would declare B as positive to SARS-CoV-2.

Replay attack against DP-3T systems. According to [DP-20], the low-cost design of DP-3T is subject to replay attacks occurring within a day. Differently to the low-cost design of DP-3T, the GAEN environment uses an ephemeral identifier that is valid for 24 hours and mitigations are proposed in [Vau20a] and [Pie20a]. On the other side, since in the unlinkable design, ephemeral identifiers are

cryptographically linked to the epoch in which they are broadcast, it is possible to make a replay attack only in an epoch.[18]

Differently to the low-cost design of DP-3T, the GAEN environment uses a secret key that is valid for 24 hours and, as stated in the documentation of GAEN [App20b], the system is vulnerable to replay attacks in a window of 2 hours.

6.7 Pronto-C2: Design and Analysis

One of the main drawbacks of previous solutions, in particular in DP-3T [DP-20] and MIT-PACT [PAC20] systems (in all their variants), is the possibility for an attacker to test whether a set of pseudonyms belongs to the same infected person and thus to infer the victim's movements. The problem is evident in the low-cost DP-3T system but, as analyzed in Section 6.6, also arises in the DP-3T's unlinkable variant.

Our approach diverges radically from the one of the DP-3T systems in that we turn the paradigm upside down. In our system it is the infected person in charge of publishing data directly to people with whom he/she got in touch. It is up to each participant to verify the occurrence of a risk. This is done through careful use of cryptography, still maintaining the system practical. In this section, we present our protocol Pronto-C2. In the description of Pronto-C2 we will not explicitly deal with the use and the need of anonymous channels; these details will be taken in account in Section 6.8.

6.7.1 Pronto-C2

Pronto-C2, a brief overview. In a nutshell, Pronto-C2 works as follows. We assume the generator g of an elliptic curve group of prime order to be known to all participants. For simplicity, we will describe our scheme using a server Server that manages a bulletin board accessible to all participants. As explained in Section 6.1, our design is flexible, we can have blockchains or just servers depending on the desired level of transparency and performances.

Periodically, each user U performs the following update operation. Let i be the current time slot. U setups a set of ephemeral and secret keys $(\mathsf{Eph}_{\mathsf{U},i+j} = g^{\mathsf{sk}_{\mathsf{U},i+j}}, \mathsf{sk}_{\mathsf{U},i+j}), j = 0, \ldots, n-1$ for some parameter n. For $k = i, \ldots, i+n-1$, U sends to Server the string $\mathsf{Eph}_{\mathsf{U},k}$ and privately stores the address $\mathsf{addr}_{\mathsf{U},k}$ in which $\mathsf{Eph}_{\mathsf{U},k}$ appears on the bulletin board. The idea is that these addresses will be used for the next n time slots. Each n time slots U runs again the update operation; previous pairs of ephemeral and secret keys are not overridden.

At each time slot i, user U proceeds as follows. U broadcasts addr_i and listens for addresses sent by other users. Each address received can be recorded along with auxiliary information.

Consider a simple scenario in which Bob is declared infected for COVID-19 by a medical laboratory and moreover he has been in close proximity to his neighbor Alice at time i (among possibly many other contacts). Let us denote by $\mathsf{Eph}_\mathsf{A} = g^{\mathsf{sk}_\mathsf{A}}$ (resp., $\mathsf{Eph}_\mathsf{B} = g^{\mathsf{sk}_\mathsf{B}}$) Alice's (resp., Bob's) ephemeral key at the time of the contact. Bob computes $K' = \mathsf{Eph}_\mathsf{A}^{\mathsf{sk}_\mathsf{B}}$ and uploads to Server the "key" (or "call") $K = H(K'||\mathsf{Eph}_\mathsf{B}||\mathsf{Eph}_\mathsf{A})$ after requiring some authentication service AuthService to blind sign K. We require signatures by the authentication service to prevent DoS attacks, and we use blind signatures to prevent the government to link patients to information on the server. To perform the authentication

[18] The DP-3T documents suggest 15 minutes as length of an epoch, so concluding that the replay attack can be performed only in a same length window. In our opinion this is imprecise due to the following reason. To preserve privacy, the length of the epochs should be randomized otherwise trivial tracing attacks can be carried out. Therefore, the epoch should not be exactly 900 seconds but a random number of seconds between, e.g., 900-d and 900+d minutes where d is a given bound (e.g., 60 seconds). Taking in account such randomization, the n-th ephemeral identifier derived by the secret key will be used up to $n \cdot d$ seconds later than the case without randomization. So, to preserve privacy the epoch length needs to be randomized and in this case the window in which the unlinkable design is secure against replay attacks turns out to be hours, not few minutes. Notice that randomization of epochs is concretely implemented in GAEN, probably for the same reason, and in fact GAEN is vulnerable to replay attacks in a window of 2 hours.

Bob needs to send to AuthService an *activation code* that Bob received from the laboratory when he got the diagnosis. We assume that Server accepts only keys with valid signatures.

At the end of the day, if Alice wants to know whether she has been in contact with an infected person, she does the following. For each address she received from a nearby user, she retrieves from Server the corresponding ephemeral key so she has Bob's ephemeral key Eph_B. She computes $K' = \mathsf{Eph}_\mathsf{B}^{\mathsf{sk}_\mathsf{A}}$ and $K = H(K'||\mathsf{Eph}_\mathsf{B}||\mathsf{Eph}_\mathsf{A})$, downloads from Server the recent keys and then searches for occurrences of K in the downloaded keys. If K is present she is notified the risk.

As an additional step, to avoid DoS attacks at the moment in which users store their ephemeral keys to the bulletin board, we add a further authentication step: each user U that wants to store the ephemeral keys to the bulletin board must contact AuthService and obtains a blind signature for each of the ephemeral keys that U wishes to store. This step will leak information to AuthService: AuthService will know which person is using Pronto-C2 and which one is not using it. It is possible to mitigate this issue by requiring persons not using Pronto-C2 to request to AuthService blind signatures for dummy ephemeral keys.

Pronto-C2's system and crypto ingredients. The ingredients of our system are:

- A secure elliptic curve group of prime order p. We assume a generator g of the group to be publicly known to all participants.
- A blind signature scheme. The blind signature is used only to authorize an authentication service managed by the government to sign user's data while hiding the message. We refer to [Cha83, Cha88] for the syntax and security properties of blind signatures.
- A server Server that is used as a bulletin board (see previous discussion and Section 6.1). The server allows any user to write data of the type "ephemeral keys" and "key", if the user is able to provide a valid (blind) signature issued by the authentication service. Keys will be written on the server only if the signature is valid.
- We assume the smartphone application has the capability to communicate with Server in an anonymous manner, hiding the real identity of the user. In Section 6.8 we show how to implement this functionality.

Pronto-C2's setting and actors. The actors involved in our protocols are:

- The users who run a smartphone application endowed with a BLE identifier beacon. A generic user will be denoted by U.
- The server (Server) that manages the bulletin board.
- A set of medical laboratories (HAs) who can engage with users in medical examinations and tests for the virus and release the activation codes to users (see below).
- The authentication service (AuthService) that is used by all users to be authorized to upload the ephemeral keys on the bulletin board and is used by an infected U to get authorization to write data about contacts on the bulletin board. AuthService releases a set of random activation codes to each HA. User U is handed an activation code Code from HA when tested positive and U can later use Code to request a signature on some data K to AuthService. The authentication service will blind sign K only if Code is a valid authentication code released by AuthService. U can then use the signature to upload K to Server.

– U: configure the smartphone application and set the time slot to 1.

– Server: perform any necessary step to accept incoming read and write requests.

– AuthService: publish two public keys for a blind signature scheme, one to be used by the users to request an authorization to upload ephemeral keys and one to request an authorization to upload shared keys. In addition, choose random activation codes and distribute a set of them to each HA.

– HA: receive a set of activation codes from AuthService.

Figure 6.1: Setup procedure.

The Pronto-C2 system. Each user U keeps a set P_{U} that is empty at the onset. Moreover, U keeps an internal variable called *time slot*. At the start of the protocol U's time slot is set to 0 and each X seconds the time slot is increased by 1. X is a parameter of the protocol (e.g., 300 seconds).

We describe Pronto-C2 through the following procedures and events.

- Setup procedure. Each actor runs a setup as described in Figure 6.1 when joining the system (i.e., the setup procedure is not a pre-processing performed simultaneously by all players before next steps).
- Update procedure. This procedure, described in Figure 6.2, is run periodically by each user U each n time slots (i.e., when U is at time slot j and j is a multiple of n).

  We assume each time slot to be short enough to prevent significant linkage of ephemeral keys to users' movements, but long enough to correctly evaluate exposure risks. Moreover, we assume the parameter n to be sufficiently large to not require the users to perform the expensive Update procedure too frequently (e.g., n can be set so that the update is performed each week).
- Broadcast procedure. There is a Broadcast procedure, described in Figure 6.3 that is run multiple times within the time slot. The frequency with which this procedure is executed within a single time slot is another parameter of the protocol.
- Listen Event. The Listen Event, described in Figure 6.4, is triggered when a BLE identifier beacon is received.
- Test Positive Event. The Test Positive Event is triggered when a user tests positive for SARS-CoV-2 at one of the laboratories of one of the HAs. When a user U gets a positive result for SARS-CoV-2 at HA's lab, U gets from HA an activation code Code. After the test (and possibly during a certain number days), U chooses a subset P'_{U} of P_{U}. U can decide upon which time slots to insert in P'_{U} based on any arbitrary criteria (e.g., can exclude time slots in which U suspects to have met some people to whom he wants to hide his disease) and interacts with AuthService to get a blind signature and then perform an upload to Server.

  More in details, when the event is triggered, U interacts with Server and HA as depicted in Figure 6.5.

In the Update procedure executed at time slot i, each user U interacts with Server and $\mathsf{AuthService}$ as follows.

- $\mathsf{U} \rightarrow \mathsf{AuthService}$: for each $j = 0, \ldots, n-1$ generate a pair of ephemeral and secret keys $(\mathsf{Eph}_{\mathsf{U},i+j} = g^{\mathsf{sk}_{\mathsf{U},i+j}}, \mathsf{sk}_{\mathsf{U},i+j})$ drawing an element $\mathsf{sk}_{\mathsf{U},i+j}$ at random from $\mathcal{Z}_p$.[a] For each $j = 0, \ldots, n-1$ interact with $\mathsf{AuthService}$ to obtain a blind signature of $\mathsf{Eph}_{\mathsf{U},i+j}$.
- $\mathsf{U} \rightarrow \mathsf{Server}$: for each $j = 0, \ldots, n-1$ upload $\mathsf{Eph}_{\mathsf{U},i+j}$ to Server after showing the corresponding blind signature computed at the previous step and store the address addr_{i+j} in which $\mathsf{Eph}_{\mathsf{U},i+j}$ appears on the bulletin board.

HAs do not perform any operation.

[a] To optimize the space, the user could choose a single seed s during the Setup procedure and in each time slot i derive $\mathsf{sk}_{\mathsf{U},i} = \mathsf{PRF}(s, i)$.

Figure 6.2: Update procedure.

- U: Let i be the current U's time slot. Broadcast the address addr_i generated in the last Update procedure using BLE.

Other participants (HAs, Server and $\mathsf{AuthService}$) do not perform any operation.

Figure 6.3: Broadcast procedure.

When a BLE message is received as consequence of a broadcast procedure, the Listen Event is triggered by the user U that receives the message and proceeds as follows.

- U: let addr_R be the address contained in the received message, i the current time slot and t any other auxiliary information (e.g., BLE signal, location, time).

  Add $(\mathsf{Eph}_{\mathsf{U},i}, \mathsf{sk}_{\mathsf{U},i}, \mathsf{addr}_R, t)$ to the set P_{U}, where $\mathsf{Eph}_{\mathsf{U},i}$ (resp., $\mathsf{sk}_{\mathsf{U},i}$) is the ephemeral key (resp., secret key) that U computed in the last Update procedures.

Other participants (HAs, Server and $\mathsf{AuthService}$) do not perform any operation.

Figure 6.4: Listen Event.

– Interaction between U and HA: once U is tested positive at HA, U gets from HA an activation code Code to interact with AuthService.

– U $\xleftarrow{\$}$ Server: (at any time after the positive test or during some given time window) choose a subset P'_{U} of P_{U} and for each quadruple $(\mathsf{Eph}_{\mathsf{U}}, \mathsf{sk}_{\mathsf{U}}, \mathsf{addr}_R, t) \in P'_{\mathsf{U}}$, retrieve from Server the ephemeral key Eph_R stored at address addr_R, compute $K' = \mathsf{Eph}_R^{\mathsf{sk}_{\mathsf{U}}}$ and $K = H(K'||\mathsf{Eph}_{\mathsf{U}}||\mathsf{Eph}_R)$ and add K to K, where K is the set of all keys that U wants to store on Server. Next, do the following:

  * Interaction between U and AuthService: for each value $K \in \mathsf{K}$ computed by U as before, U uses its activation code Code to interact with AuthService to compute a blind signature σ of K.
  * U $\rightarrow$ Server: for each $K \in \mathsf{K}$ computed by U as before, send K and σ to Server.
  * Server $\xleftarrow{\$}$ U: upon receiving any pair (K, σ) from U, verify σ and if the signature is valid add K to the bulletin board.

Figure 6.5: Test Positive Event.

- Verify procedure. This procedure, described in Figure 6.6, is carried out by a user U who wants to discover whether she got in contact with some other user U^+ who tested positive for SARS-CoV-2.

6.7.2 Analysis of Pronto-C2

In this section, we informally argue that Pronto-C2 withstands all the attacks shown in Section 6.4 and 6.5. For the sake of simplicity for all attacks except Paparazzi attack, we analyze Pronto-C2 as if ephemeral keys were directly sent over the BLE channel, ignoring the use of addresses. We therefore assume that the procedure to store ephemeral keys on the bulletin board effectively hides the real identity of the owner of each key.

- Paparazzi attack (cfr., Section 6.4.1):
  Recall that this attack assumes the attacker Adv to use only passive devices which operate in reception mode and are not able to transmit any signal. The only information a passive device D observes consists of ephemeral keys exchanged by users at the position in which D is located.

  To track a user of the system the attacker Adv can do what follows.

  Adv can try to link together the ephemeral keys used by the target user U with the ones recorded using passive devices. Knowing which ephemeral keys belong to U, Adv could easily track U. This attack is not applicable to Pronto-C2 since different ephemeral keys generated by the same user U are not linkable.

  Adv could try to track a target user U exploiting the calls available on the bulletin board. In this case the target of the attack is an infected user. Since the devices used by Adv are passive, no calls of U will ever be directed to a device D controlled by Adv. The only way for Adv to track U is to extract the ephemeral keys used to generate the calls and associate them to a single user. Since the calls are anonymously sent to Server, it is impossible for Adv to link together the calls

When a user U wants to verify whether she got in contact with any user U^+ who got a positive result for SARS-CoV-2, U engages in an interactive protocol with Server as follows.

- $\mathsf{U} \xleftarrow{\$} \mathsf{Server}$: Let P_{U} the set computed by U during the protocol execution so far. For each quadruple $(\mathsf{Eph}_{\mathsf{U}}, \mathsf{sk}_{\mathsf{U}}, \mathsf{addr}_R, t)$ in P_{U} do the following:
  - Retrieve from Server the ephemeral key Eph_R located at address addr_R. Compute $K' = \mathsf{Eph}_R^{\mathsf{sk}_{\mathsf{U}}}$ and $K = H(K'||\mathsf{Eph}_R||\mathsf{Eph}_{\mathsf{U}})$, download the recently uploaded keys from Server and search for K.[a] If K is present, compute the risk and notify U.

HAs and $\mathsf{AuthService}$ do not perform any operation.

[a] As we described the protocol, the user does not directly check the signature since the validity of the signatures is checked when the keys are uploaded to Server. For a stronger verifiability guarantee we can change the protocol so that the user is given the possibility to download and check the signatures.

Figure 6.6: Verify procedure.

of U. However, the analysis of Orwell attack reported below shows that, even linking all the calls, it would be impossible for Adv to extract the ephemeral keys related to each call.

- Orwell attack (cfr., Section 6.4.2):
  The attack differs from Paparazzi attack in the fact that the adversary Adv can collude with $\mathsf{Server}, \mathsf{AuthService}$ and HA. In this scenario, Adv has the following additional advantages with respect to the attacker of Paparazzi attack:
  - Adv knows the real identities of the infected users and the associated code Code assigned to them by HA to be authorized to obtain the blind signatures for the calls;
  - Adv knows the blinded messages that the infected users asked $\mathsf{AuthService}$ to sign.

  This information is not useful for Adv to track a user U. Indeed, to track U, Adv needs to link all the calls published by U with U's ephemeral keys. Assuming that the upload of the calls on the bulletin board is performed through an anonymous channel, in order to link these calls, Adv needs to discover which calls were blinded by U to obtain the corresponding signature from $\mathsf{AuthService}$. This would require Adv to break the blindness property of the blind signature, that is unlikely for a polynomial adversary. Even if Adv broke the blindness of the signature scheme, the additional information received is the set of calls sent to Server by U, but none of them can be a call that Adv can understand, since none of passive devices controlled by Adv is the recipient of a call. Then, Adv would need to take all the couples of ephemeral keys recorded by each passive device D_i, and try to compute a call between the two users that owns these two ephemeral keys. If the computed call is equal to a call published by U, then Adv knows that U was located in proximity to D_i. By doing it for each passive devices, Adv could track the movements of U in the last 14 days. However, if Adv is able to successfully compute a call starting from two ephemeral identifiers, it is easy to show that Adv can be used to define an adversary breaking the computational Diffie-Hellman assumption.
- Matrix attack (cfr., Section 6.4.3):

The level of resilience of Pronto-C2 to this attack can be shown with a similar argument to the one presented above for the Orwell attack. Indeed, Adv cannot link U's calls in different time slots since U uses different and unlinkable pseudonyms that in turn will produce unlinkable DH keys. Obviously there are some inherent leaks in extreme situations like for instance when there is only one new infected person in the town where pseudonyms have been collected by the adversary. Such leaks are inherent, instead the most general case with multiple new infected citizens is nicely protected by Pronto-C2 since Adv will not know which specific call was made by which infected user. Obviously the actual location related to the different calls may act as another information to disambiguate the path of an infected user, but again this is inherent and with Pronto-C2 the adversary gets lost as soon as there is some ambiguity.

- Bombolo attack (cfr., Section 6.5.1):
  Co-location information among infected users are not leaked since an infected user A will upload to Server a call of the form $K_{\mathsf{A}} = H(\mathsf{Eph}_{\mathsf{B}}^{\mathsf{sk}_{\mathsf{A}}}||\mathsf{Eph}_{\mathsf{A}}||\mathsf{Eph}_{\mathsf{B}})$ if A passed nearby B, likewise if B is an infected user B will upload the key $K_{\mathsf{B}} = H(\mathsf{Eph}_{\mathsf{A}}^{\mathsf{sk}_{\mathsf{B}}}||\mathsf{Eph}_{\mathsf{B}}||\mathsf{Eph}_{\mathsf{A}})$ if B passed nearby A. Then, the calls K_{A} and K_{B} uploaded by A and B are different and it is hard to "co-locate" these keys[19]. On the contrary, Pronto-C2 exposes the number of calls that an infected user U does when U sends these calls to the AuthService. We note that this leak of information can be mitigated adding some dummy calls.

  We propose this attack because the unlinkability of the calls introduced by uploading $K = H(K'||\mathsf{Eph}_{\mathsf{A}}||\mathsf{Eph}_{\mathsf{B}})$ instead of just K' contradicts the message that DP-3T's risk analysis (SR 6) [DT20b] seems to convey when claiming that "an infected person uploads all identifiers observed during the contagious window to the server. For epochs in which groups of at least three people were in close proximity to each other, this will reveal temporal colocation information about infected individuals to the server.".

- Brutus Attack (cfr., Section 6.4.4):
  The data uploaded by a user U in Pronto-C2 cannot be linked to the real identity of U. Data are uploaded in the following steps:

  1. when the infected user uses Code to access AuthService in order to obtain the blind signatures of the calls and
  2. when the infected user uploads the calls to Server along with the unblinded signatures.

  The first step involves uploading Code to AuthService in order to obtain the blind signature of the calls. Since HA knows the real identity of each infected user, it is possible for Adv to link the blind signature requests with an infected person. However, since the upload of the calls is performed through an anonymous channel, Adv cannot link the calls with the signature requests thanks to the blindness property of the signature scheme. This of course hides the authorship of the uploaded data only inside the set of the infected users, which is known to Adv.

- Gossip attack (cfr., Section 6.5.2):
  At first sight, one could think that a proof of contact with an infected user U can be given by user A providing a proof about the calls on the bulletin board. For instance, let A be a user holding a secret key sk_{A} corresponding to Eph_A and let $\mathsf{Eph}_{\mathsf{U}}$ be the ephemeral key of a user U. If A finds a call $K = H(\mathsf{Eph}_{\mathsf{U}}^{\mathsf{sk}_{\mathsf{A}}}||\mathsf{Eph}_{\mathsf{U}}||\mathsf{Eph}_{\mathsf{A}})$ on the bulletin board, A could prove that he knows the secret key sk_{A} corresponding to $\mathsf{Eph}_{\mathsf{A}}$ and that K is computed as before, thus proving that U made a

[19] We note that this analysis shows that co-location is difficult to obtain for both the regular users and the server that colludes with the health authority.

call to A. (Notice that the only data needed by A to prove a contact with U is the secret key sk_A.) However, this alleged attack does affect Pronto-C2. Let us elaborate on that.

Recall that in Pronto-C2 all the pseudonyms used by the users are made public on the bulletin board. So, if A has never been in contact with U, A could use Eph_U, that is public on the bulletin board, to compute a call $K = H(\mathsf{Eph}_\mathsf{U}^{\mathsf{sk}_\mathsf{A}}||\mathsf{Eph}_\mathsf{U}||\mathsf{Eph}_\mathsf{A})$ (a call from U to A) and show sk_A as proof of the fact that such call has been done. Generally, any proof of the fact that U made a call to A is *not* evidence of the fact that U met A since such proof could have been computed by A even if U has never been in contact with A.

Notice however, that in the case that A is not infected, a proof of the fact that U made a call to A is instead plausible evidence of the fact that U met A: indeed, only infected users can write calls to the bulletin board and U is honest. (If U is dishonest, the pair U and A can be seen as a single adversary.) For this reason, we say that the attack affects Pronto-C2 minimally in the sense that an attacker A can provide a proof (that, as shown before, is the secret key sk_A) of the contact between U and A that convinces a third party B who believes that A is not infected. Using this fact, we can interpret this attack as a *feature*. Indeed, similarly to what stated in Section 6.5.2, laboratories could give a higher priority to users who are able to provide such a plausible evidence of having met an infected individual. This feature could be also very useful in assuring that reliable data about encounters among infected individuals are provided to epidemiologists.

- Matteotti attack: (cfr., Section 6.5.3).
  Every key K stored on the bulletin board has the form $K = H(K'||\mathsf{Eph}_\mathsf{TSen}||\mathsf{Eph}_\mathsf{B})$. A user B who at some time t broadcasts Eph_B will be notified a risk only if B received at time t an ephemeral key $\mathsf{Eph}_\mathsf{TSen}$ and $K' = \mathsf{Eph}_\mathsf{TSen}^{\mathsf{sk}_\mathsf{B}}$. Since it is hard for Adv to compute K' without knowing sk_B or $\mathsf{sk}_\mathsf{TSen}$, we conclude that B is alerted only when B actually met TSen and TSen put an alert for B. However, in such case the alert corresponds to an actual risk for B and does not represent a successful attack.

- Replay attack: (cfr., Section 6.5.4).
  Pronto-C2 is not subject to replay attacks. An adversary Adv who broadcasts, at location X, the pseudonym of a user U_1 collected during a prior time slot in a different location Y, would fail in the attempt of causing false at-risk notifications. Indeed, to be notified, a user U_2 needs to find, on the bulletin board, a call which is directed to himself and is generated by the infected user U_1. Since the generation of such call requires the secret key of U_1 related to the time slot when the alleged meeting took place, it would be computationally infeasible for Adv to trigger a fake at-risk notification for U_2.

  Similarly, it is easy to see that Pronto-C2 is secure against relay attacks that are one-way, that is in which the adversary can only forward information from a location X to a location Y of the space (but not from Y to X).

6.8 Suggestions for a Practical Realization of Pronto-C2

In this section we suggest a practical implementation of Pronto-C2 and analyze its performance in a real-world scenario. In the following section, Pronto-C2 will involve the following actors:

- the user U, who runs the smartphone application;
- the server Server, that manages the bulletin board;
- the medical laboratory HA;

Attacks	Low-cost DP-3T	Unlinkable DP-3T	Pronto-C2
Paparazzi	✗	✓	✓
Orwell	✗	✗	✓
Matrix	✗	✗	✓
Bombolo	✓	✓	◯
Brutus	✗	✗	✓
Gossip	✗	✗	◯
Matteotti	✓	✗	✓
Replay	✗	✓	✓

Figure 6.7: We show which system is susceptible to which attack. ✗ denotes that the system is vulnerable to the attack, ✓ denotes safety against the attack, ◯ denotes almost safety against the attack in the sense that the system is only very mildly affected (cfr., Section 6.7.2), and † denotes that a given protocol offers *solitary protection* as described in Section 6.3. Recall that Gossip attack can be turned into a feature in Pronto-C2, but not in DP-3T systems.

- the authentication service AuthService.

There is a risk for U because of linkability/deanonymization attacks due to timings and IP addresses of the TLS connections with Server when uploading or downloading data. Such attacks also affect the DP-3T designs that seems to ignore them, and in general are applicable to any system if no specific countermeasure is used. One might consider onion routing and mix networks to protect U against such attacks, but the impact on performance remains unclear. In order to give a fair description of a practical realization of our solutions, we do not ignore this issue and we therefore include in this section a mitigation based on mixers. We will consider a setting where U can freely select a mixer MixServer that she trusts, and mixers do not need to be approved by the government, they can be spontaneously run by anyone.

Server owns a pair of private and public keys $(\mathsf{sk}_{\mathsf{Server}}, \mathsf{pk}_{\mathsf{Server}})$ of a public key encryption scheme (e.g., ElGamal [Gam85] instantiated on the elliptic curve used for the key exchange), the public key of Server is made publicly available at set-up time. Every time U has to send data to Server, U will actually encrypt the data with $\mathsf{pk}_{\mathsf{Server}}$ and send the resulting ciphertexts to MixServer. A mixer waits for enough data to be collected, and then performs a mixing and sends them to Server. In addition, MixServer can also download all data from Server so that U can use MixServer also to retrieve anonymous calls and ephemeral keys.

There can be several heterogeneous mixers available, provided by large institutions like no-profit organizations, political parties, national/state/local governments, as well as several smaller mixers that can serve a district, a school, a group of friends/relatives. U will obviously choose the one that he trusts more in performing properly the service with a sufficiently large amount of collected data and without abusing it. It remains possible for a user to ignore this suggestion and to just to use some proper timing and then sending the encrypted data directly to Server using onion routing and/or relying on the partial hiding provided by mobile operators and public Wi-Fi networks since somehow they can also be seen as light forms of mixers. We will continue our discussion considering the case of a citizen using a mixer that she trusts and that remains uncorrupted.

Server works as a bulletin board, so all data ever received by Server are made publicly available after being decrypted. HAs are the laboratories that perform the SARS-CoV-2 tests. If a user U gets tested positive, a HA hands U an authorization code Code. AuthService is the service in charge of authorizing users to upload anonymous calls to the bulletin board. Moreover, in Pronto-C2, it can also be useful to

issue credentials to upload ephemeral keys to the bulletin board, in order to mitigate DoS attacks. In Pronto-C2, we instantiate H using SHA256 [NIS02] as hash function.

6.8.1 Pronto-C2 Practical Implementation

Here we discuss the main operations performed by users in Pronto-C2 and the recommendations that users should follow.

- U generates a set of 96 ephemeral and secret keys to be used for the next day (a pair every 15 minutes).
- U connects to AuthService in order to prove to be a legitimate user of the system needing to announce new pseudonyms. With this connection, U obtains 96 blind signatures of the generated ephemeral keys. This step is needed to avoid DoS attacks on the bulletin board.
- U uploads the ephemeral keys to the bulletin board. U encrypts the 96 ephemeral keys, along with the related unblinded signatures, and sends the resulting ciphertexts to MixServer that, after having collected a sufficiently large amount of data, mixes and sends them to Server who will eventually decrypt and publish them. U will obtain the addresses of his ephemeral keys by querying Server with the first l bits[20] of each key. Server will return, for each query, all the ephemeral keys that match these bits, along with the addresses of such keys. U will store the addresses of his ephemeral keys and will broadcast them during the following day. By doing so, U is able to efficiently retrieve his addresses while hiding the link between them to Server since each query corresponds to a fairly big set of ephemeral keys. To add more noise, dummy queries may also be performed.
- U downloads from Server (this step could also be performed through MixServer that can have a local copy of data available on Server) the ephemeral keys $\mathsf{Eph}_{\mathsf{U}_i}$ for all addr_i collected during the day by querying Server (or MixServer) with the first l bits of each address collected during the day. For each query, U will receive a set of ephemeral keys and U can select the needed key.
- U downloads from Server (as above this step could also be performed through MixServer) all the calls.
- If U is positive to SARS-CoV-2, U receives Code from HA who delivered him the diagnosis.
- The infected user U computes the anonymous call for each ephemeral key received.
- The infected user U encrypts the calls along with the unblinded signatures, with $\mathsf{pk}_{\mathsf{Server}}$. Then, U uploads these data via MixServer, as done for the upload of the ephemeral keys. If U feels uncomfortable in giving evidence of being infected to MixServer, U can send dummy encrypted calls on a daily basis (i.e., whenever U uploads new ephemeral keys).

We note that following our recommendations, Pronto-C2 will behave exactly as in both DP-3T designs regarding data sent during the day over the BLE channel (e.g., it will send and receive the same amount of data without performing additional computations).

[20] The parameter l will be discussed later.

6.8.2 Performance Analysis

To give an idea of the overall performance, we report an example of a concrete execution in a typical scenario. To analyze the performance, we take into account the memory usage of the smartphone application, the amount of uploaded and downloaded data, and the number of exponentiations the smartphone has to execute.

In the following examples we assume that:

1. each user U has 100 contacts per day on average[21];
2. we assume that there are on average 5000 new infected individuals per day within a single country[22];
3. U uses a new pseudonym every 15 minutes;
4. the contagion time window is 10 days long[23];
5. the dummy calls produced by the user are 100 per day on average;
6. only for Pronto-C2, we consider a scenario in which there are 5 million users that upload their ephemeral keys each day and that l is equal to 17. By doing so the resulting set of ephemeral keys would be of about 3663 elements on average. [24]. In the same scenario, if l is equal to 10, the number of ephemeral keys that is downloaded by each user to compute the anonymous calls is 468750 on average, while, increasing l to 25 the number of ephemeral keys downloaded is 15. In general, the average number of downloaded keys is the total number of ephemeral keys published divided by 2 to the number of bits fixed in the prefix. Compared to l equal to 15, in case l is equal to 10 the requests of the user are hidden in a larger number of calls, while in case l is equal to 25 the number of ephemeral keys downloaded by U is smaller and it is easier for an adversary to guess which ephemeral key was searched by U.

6.8.3 Performance Analysis of Pronto-C2

Memory consumption. Every smartphone has to maintain on the local memory all the secret keys and the ephemeral keys to be used the next day, and all the shared keys and ephemeral keys associated to the people encountered during the last 10 days. This requires $(32 + 33) \times 96 \times 11 = 68KB$ for the secret keys and the ephemeral keys, $100 \times 10 \times 32 = 32KB$ for the shared keys and $33 \times 96 \times 10 = 31KB$ for the ephemeral keys received in the last 10 days. Indeed, 33 is the size in bytes of the the compressed representation of an ephemeral key and 32 is the size of a secret key and of a shared key (each shared key is 32 bytes long since there is no need to reconstruct the curve point later on). In addition, all pseudonyms to be transmitted the next day occupy $96 \times 16 = 2KB$, where 16 is the size in bytes of each pseudonym addr_i.

We did not count the space required by blind signatures since they are sent to MixServer and erased as soon as they are received.

[21] Recall that this is an average value that takes very roughly into account that there are citizens staying at home for the entire day, areas with poor public transport, social distancing and several other factors that, at least in some countries, make this expectation quite conservative. It is anyway easy to adjust the analysis using different values.

[22] This is also a very conservative average for many countries according to current statistics.

[23] We use 10 days as contagion time window following the assumptions in [CKL+20].

[24] In practice, the smartphone could decide l dynamically, taking into account the number of ephemeral keys that have been added that day. By doing so, it can ensure that the size of the returned set of keys allows for an efficient search while providing a sufficient level of anonymity

Amount of downloaded data. The vast majority of downloaded data comes from the anonymous calls.

There are four other phases in which the user will download data:

1. to obtain the addresses of the ephemeral keys stored on the bulletin board;
2. to obtain the ephemeral keys corresponding to the addresses collected during the day;
3. to obtain the blind signatures that authorize the user to store the ephemeral keys on the bulletin board and
4. to obtain the blind signatures that authorize the user to store the calls on the bulletin board when positive to SARS-CoV-2.

If Pronto-C2 is instantiated using blind signatures based on RSA [BNPS03] with a length of 2048 bits for the modulus, the overall data exchanged to obtain a signature corresponds to $256B$, therefore, the only significant steps in terms of downloaded data are 1 and 2.

To obtain the addresses of all ephemeral keys stored on the server, the data downloaded are $3663 \times 33 \times 96 = 12MB$. The same applies to obtain the ephemeral keys corresponding to the addresses collected during the day.

Amount of uploaded data. The amount of uploaded data depends on the amount of anonymous calls.

In addition, in Pronto-C2, each user uploads to Server every day the ephemeral keys to be used the next day. The amount of data uploaded is $33 \times 96 = 4KB$. Moreover, each user uploads 96 blind signatures to AuthService, that means uploading $256 \times 96 = 24KB$ and each user will upload every day 1000 dummy calls on average, that means $281KB$ per day.

Number of exponentiations. Pronto-C2 uses a blind signature scheme based on RSA to generate the signatures and an ElGamal encryption scheme to encrypt the messages.

We assume that every day each user computes 1000 dummy calls, computing 2000 exponentiations.

Every time U uploads an ephemeral key on Server, U needs to obtain a blind signature by AuthService and must encrypt the ephemeral key. To get such a signature, U needs to compute 1 exponentiation. To encrypt the ephemeral key and the related signature U computes 9 more exponentiations, in addition to the one for the ephemeral key itself. So the overall process requires 11 exponentiations.

Since the ephemeral keys to be uploaded each day are 96, the total number of exponentiations needed to store all the ephemeral keys is $96 \times 11 = 1056$ per day.

Every time an infected user U uploads his set of anonymous calls, U needs to obtain a blind signature for each of these calls and must encrypt the call along with the related signature. Then, the total number of exponentiations is $(10 \times 1000) = 10000$[25].

Checking the risk of contagion requires 1 exponentiation to compute the shared key for each pseudonym encountered during the day, since the shared keys of the encounters of the previous days are already stored. The total number of exponentiations is 100.

Therefore, the total number of exponentiations computed on average per day by a smartphone of a non-infected user is $1056 + 100 + 2000 = 3156$. If the user is tested positive to SARS-CoV-2, the number of additional exponentiations he performs is 10000.

[25] We remark that, there is no need for the user U to compute the shared key to put in the call since U computes the shared keys from the pseudonyms encountered day by day and stores these data for 10 days.

Improved performance. We now consider a concrete instantiation which is resilient to all attacks as discussed in the analysis of Pronto-C2 except for Bombolo and Brutus attacks. Indeed if one is protected by Paparazzi attack and Orwell attacks, then we notice that the impact of a successful Bombolo or Brutus attack is much milder (the same does not hold for DP-3T designs since they are insecure w.r.t. Orwell attacks). We show that under this relaxed but acceptable privacy, still way better than what is achieved by the DP-3T designs, one could obtain consistent performance improvements. For instance, an ACT system which is vulnerable to both Brutus and Orwell attacks is in turn vulnerable to link users' movements to their real identities. On the other hand, the Brutus attack on its own just leaks the ownership of the data uploaded by an infected user, which in Pronto-C2 are computationally indistinguishable from random strings [26]. Therefore, taking this trade-off into account, one could also opt for improving the overall performance as follows:

- Batching all the anonymous calls in a single signature request to AuthService. This would leak the number of contacts an infected user had, but would dramatically reduce the number of exponentiations.
- Uploading the authorized calls directly to Server without the intervention of MixServer. This would simplify the upload procedure, at the price of exposing infected users to the Brutus attack via IP address linking, if Server and AuthService collude.

[26] Strictly speaking this is true only for a distinguisher who is not the recipient of a call. If one is the recipient of a call, she can of course discriminate a call from a random string and evaluate her risk of contagion.

Chapter 7

Smart Contracts Realizing the Terrorist Attack to GAEN

In this chapter, we show that an adversary can attack the integrity of contact tracing systems based on Google-Apple Exposure Notifications (GAEN) by leveraging blockchain technology. We show that through smart contracts there can be an on-line market where infected individuals interested in monetizing their status can upload to the servers of the GAEN-based systems some keys (i.e., TEKs) chosen by a non-infected adversary. In particular, the infected individual can anonymously and digitally trade the upload of TEKs without a mediator and without running risks of being cheated. This vulnerability can therefore be exploited to generate large-scale *fake* exposure notifications of at-risk contacts with serious consequences (e.g., jeopardizing parts of the health system, affecting results of elections, imposing the closure of schools, hotels or factories).

As main contribution, we design a smart contract with two collateral deposits that works, in general, on GAEN-based systems. Roughly, in such smart contract both parties deposit some amount of cryptocurrency, and when the infected individual uploads a list of TEKs signed by a verification authority, the smart contracts checks if (1) the signature is correct and (2) if the terrorist TEKs, stored into the the smart contract, are present in the provided TEKs list. If such conditions are correctly verified (by the smart contract) the terrorist deposit is transferred together with the infected individual's deposit to the infected individual's wallet. Otherwise, as a punishment, the terrorist gets back his deposit and the infected individual's deposit gets burned. We then also suggest the design of a more sophisticated smart contract, using DECO, that could be used to attack in a different way GAEN-based systems (i.e., this second smart contract can succeed even in case GAEN systems are repaired making ineffective the first smart contract). The main feature of DECO is that offers a decentralized oracle certifying that a TLS communication between two parties has been established, and that certain data have been indeed transferred. By relying on this system, the infected individual can certify the smart contract that the he has correctly uploaded the TEKs that are stored in the smart contract (the terrorist TEKs) to the server's authority.

Our work shows how to realize with GAEN-based systems (in particular with Immuni and Swiss-Covid), the terrorist attack to decentralized contact tracing systems envisioned by Vaudenay.
This work is part of a paper accepted to ACNS '21 [AFV21].

7.1 Introduction

During the COVID-19 pandemic, several governments have decided to use digital contact tracing systems in addition to other practices to contain the spread of SARS-CoV-2. The reason is that digital contact tracing could help in notifying at-risk exposures to individuals that have been in close proximity to people who subsequently tested positive to SARS-CoV-2. This could be very useful especially when

the involved individuals do not know each other. If digital contact tracing systems worked perfectly, they would certainly be effective in alerting at-risk individuals who, following some prescribed procedures (e.g., informing doctors, staying at home in self-quarantine), may significantly limit the spread of the virus. Such systems have been highly recommended by some governments and in some cases (e.g., in Switzerland) an alert received by a contact tracing smartphone application allows to get a test for free.

The most used contact tracing systems rely on Google-Apple Exposure Notifications (GAEN), a feature offered by recent updates of iOS and Android and therefore available on a large fraction of currently used smartphones. These systems are widely used in Europe (e.g., Austria, Belgium, Germany, Ireland, Italy, Poland, Spain, Switzerland) and cross-border compatibility has recently been implemented[1].

Moreover, in the US, several states have adopted GAEN-based systems. GAEN allows to run decentralized contact tracing where there is very low control from governments, and this makes attacks from third parties generally simpler to mount and harder to mitigate.

GAEN-based contact tracing systems. The approach of GAEN-based contact tracing systems is to use Bluetooth Low Energy (BLE) to detect close proximity contacts among smartphones. Each smartphone broadcasts random pseudonyms via BLE, and this information is received by smartphones in close proximity along with some encrypted metadata. If a citizen is tested positive and decides to notify others, she will upload a set of secret keys named *Temporary Exposure Keys* (we will refer to them as TEKs in the remainder of the paper) corresponding to previous days in which she was presumably contagious. Starting from a TEK, it is possible to generate all the pseudonyms broadcast by a user during a day. The receivers of such pseudonyms will then manage to decrypt the stored metadata to then evaluate a risk factor[2]. The TEKs are disseminated to the users via a back-end server that periodically posts a list of digitally signed TEKs. A detailed description of GAEN can be found at https://covid19.apple.com/contacttracing.

An important point is that GAEN evaluates the reported TEKs if and only if the digital signature verifies successfully under a public key that has been previously communicated by the developers to Apple and Google. Google motivates this requirement saying that it ensures that keys received by the devices are actually from the authorized server and not from malicious third parties[3].

Theoretically, one could also rely on server authentication using TLS, but the use of Content Delivery Networks (CDNs) to disseminate TEKs (e.g., the CDN used by Immuni is operated by Akamai, while the SwissCovid's one is operated by Amazon) requires protection against malicious modifications operated by the CDN itself. Unfortunately, as we will see next, this requirement paves the way for the development of dark economies where TEKs to be uploaded by infected users are traded through smart contracts.

False positives due to attacks. Since BLE was not originally designed to detect a precise distance among devices, the evaluation of the risk factor is prone to significant errors. To this regard, Leith and Farrell recently evaluated the reliability of BLE for digital contact tracing in several real-world scenarios [LF20b].

While false positives due to BLE limitations in measuring distance can indiscriminately affect all individuals using the smartphone apps, a much more concerning threat allowing to direct false positive alerts to specific targets has been pointed out in prior work (e.g., see [Vau20a, Pie20a]). Indeed, GAEN-based contact tracing systems[4] can be heavily abused through replay attacks. In this case, the

[1] EU eHealth Network: European Proximity Tracing. An Interoperability Architecture https://lasec.epfl.ch/people/vaudenay/swisscovid/swisscovid-ana.pdf.

[2] For example, metadata include information useful to estimate the distance among the smartphones which clearly impacts on estimating the risk of a contact.

[3] Google: Exposure Notification Reference Key Server https://google.github.io/exposure-notifications-server/.

[4] Sometimes for brevity we will just say GAEN systems.

pseudonyms sent by an individual considered at risk (e.g., a person who is taking a test) are transmitted by an adversary to a different location in order to create a fake proximity contact. The attack can have a specific target but can also be performed at large scale. Recently, in [RGK20] Gennaro et al. discussed how the capability of running such attacks at large scale can be used to put a category of citizens in quarantine with the consequence of severely compromising the results of an election. In general, the malicious generation of false positives can be harmful in various ways, the health system can be overloaded of requests that can penalize those citizens who instead are really affected by the virus. A student can cause the complete closure of a school or university and similar attacks can be directed to shops, malls, gyms, post offices, restaurants, factories.

Risks related to replay attacks were already known back in April 2020, and GAEN systems have a pretty large time window (about 2 hours) [Goo20b] for pseudonyms to be replayed successfully. Nevertheless, governments have so far considered unlikely that such attacks can produce enough damage to cancel out the positive effects of genuine notifications of at-risk contacts. This could be due to complications involved in the attack. Indeed, an adversary may not want to get herself infected, or it could not be easy to identify, and be in physical proximity with, an individual that soon will report to be infected. In [Vau20b], Vaudenay envisioned the possibility of using smart contracts to realize a *terrorist* attack against decentralized systems, therefore the attack could potentially apply to GAEN-based systems as well. In this case, the attacker would spread on his targets some pseudonyms, e.g. by placing beacons close to the targets or getting close to the targets while running the digital contact tracing app on a smartphone, subsequently promising through a smart contract a reward to whoever uploads the corresponding keys. Therefore, an infected individual who participates in the contract will cash a reward, and false positive alerts will raise on the smartphones of the targets selected by the terrorist. More details are discussed in Sec. 7.1.2.

7.1.1 Our Contribution

We show that the terrorist attack envisioned by Vaudenay can be concretely mounted against currently deployed GAEN-based contact tracing systems. In particular, we have analyzed its concrete feasibility with respect to two systems, such as Immuni [Imm20], used in Italy and SwissCovid [Swi20a], used in Switzerland. We expect several other deployed GAEN systems to suffer from the same vulnerabilities.

More generally, our work shows how to attack the integrity of currently deployed GAEN-based contact tracing systems by leveraging blockchain technology. A very alarming side of our contribution is that current systems can be compromised without the need for the attacker to get infected, or to be with high probability in close proximity to individuals that will be soon detected positive and will upload the keys. Our attacks consist of smart contracts to establish a mediator-free market where parties, without knowing each other, without meeting in person and without running risks to be cheated, can abuse exposure notifications procedures of GAEN systems. We give a brief description of the mentioned smart contracts in the following.

Trading TEKs exploiting publicly verifiable lists of *infected* TEKs. As a main contribution we show a smart contract named Take-TEK that allows a buyer (i.e., the adversary willing to spread false positive alerts)to store the TEKs that the seller (i.e., the infected individual that is willing to monetize her right to upload TEKs to the servers of the GAEN system) will then upload to the server . The smart contract requires the buyer to deposit the amount of cryptocurrency (we will call it prize) that he is willing to give to the seller. The seller instead will deposit an amount of cryptocurrency in order to reserve a time slot in which she will try to upload the TEKs. In case she does not manage to complete the upload of the TEKs[5], the deposit will be assigned to the buyer. The deposit of the

[5]Notice that if the seller was infected in a period of time that can include the buyers' TEKs stored in two (or more) different smart contracts, he can serve both of them by uploading to the server the buyer TEKs stored in both the smart contract.

seller is therefore useful to make unlikely that a seller tries to prevent other sellers from completing the job. Additionally, we can hide the TEKs so that, even when observing all transactions, it is not clear which TEKs have been traded using the smart contract among the many TEKs jointly published by the server of the contact tracing system during a slot.

Take-TEK crucially relies on the server publishing such lists of TEKs along with a signature verifiable with a publicly known public key. We show that the Take-TEK attack can be deployed to generate fake false positive alerts w.r.t. both Immuni and SwissCovid. Indeed, both systems follow strongly the design of GAEN and announce such signed lists of TEKs using ECDSA signatures.

Regardless of Immuni and SwissCovid making available or not their signature public keys, we have successfully extracted the public keys from previously released signatures[6]. Therefore, Take-TEK can be instantiated to attack both (and possibly many more) systems. More details are discussed in Sec. 7.2.

Trading TEKs without publicly verifiable signatures: DECO. One might think that realizing the terrorist attack via smart contracts (e.g., Take-TEK) crucially relies on exploiting those signed lists of TEKs under a known (or extractable) public key since the smart contract needs to check the signature of the list of TEKS to ensure its integrity. At first sight, a fix to such vulnerabilities consists of hiding the public keys and to use a signature scheme such that it is hard to extract the public key from signed messages. However, we show that things are actually more complicated for designers of contact tracing systems. In particular, we show another way to buy/sell TEKs that follows a completely different approach. The key idea is requiring the seller to prove that a TLS session with the server led to a successful upload of the buyer's TEKs. The necessary requirements on the communication between smartphone app and server are that 1) both the TEKs and the positive (or negative) outcome of the upload procedure are part of the exchanged application data in the TLS session, and 2) the upload phase consists of just one request made by the client and the response of the server (e.g., as it is in SwissCovid). At first sight, the attack seems very hard to realize since notoriously TLS produces deniable communication transcripts when it comes to application data (i.e., exchanged messages are only authenticated and not digitally signed). However, we exploit a very recent work of Zhang et al. [ZMM+20]. They show how to build a fully decentralized TLS oracle, named DECO, for commonly used ciphersuites. Further details are described in Sec. 7.3.

Remark on the actual work done by our smart contracts. Both Take-TEK and the DECO-based smart contract provide full guarantees to both seller and buyer at the expense of running some cryptographic operations that can obviously produce transaction costs. Nevertheless, if we make an additional optimization based on pragmatism, the expensive computations may happen very rarely in practice. Indeed, we notice that the main computational cost for those smart contracts consists of checking at the very end that the seller has completed the task of uploading TEKs correctly. We observe that a buyer can check that TEKs are published by the server on his own. As a result, he would be satisfied in finding out that the trade has been completed successfully. Therefore, it is natural to expect that the buyer would give his approval to the smart contract to transfer the money to the seller avoiding the execution of expensive computations, and therefore saving transaction costs[7]. Since this behavior would be visible in the wild, the reputation of the buyer would also benefit from such approvals and more sellers would want to run contracts with him. Moreover, a (somewhat irrational) buyer that refuses to speed up the execution of the smart contract would anyway not stop the final

[6] In GAEN-based systems, signature verification is crucial to ensure data integrity. Our smart contract needs to check the signature to be sure that the seller did not sent corrupted data.

[7] Obviously, the smart contract can be adjusted so that, in case the buyer does not give his approval and the seller shows that she completed successfully her part of the contract, the expensive transactions costs due to the lack of help from the buyer are charged to the wallet of the buyer. A simple way to realize this could be asking for an additional deposit made by the buyer which could clearly cover the transaction costs of the seller in case the buyer does not give his approval and the seller shows that she successfully completed the upload procedure.

transfer of the deposited money to the seller. As a result, the buyer would only get a worse reputation. In conclusion, the expensive work done by our smart contracts belongs to pieces of code that would rarely be executed in practice.

7.1.2 Related Work

The design of GAEN is very similar to the low-cost design of DP-3T [DP-20], and thus several vulnerabilities identified in prior work generally apply to both systems. Tang [Tan20b] observes that DP-3T is vulnerable to identification attacks, and presents an accurate survey about contact tracing systems. In [Vau20a], Vaudenay reports both privacy and security issues. The most famous privacy attack is the so-called *Paparazzi* attack. Basically, it is possible through passive antennas to track infected individuals over a certain time window[8] during which pseudonyms are linkable.

Regarding security issues, Vaudenay extensively considers false alert injection attacks, where the adversary manages to raise false alerts on the smartphone apps of targeted victims. Within this category, there are *replay* and *relay* attacks. GAEN is vulnerable to relay attacks and to replay attacks carried out within two hours. Vaudenay in [Vau20a] and Pietrzak in [Pie20b] proposed, back in April 2020, some solutions to defeat these attacks, but they have not been included neither in DP-3T nor in GAEN designs so far. Baumgärtner et al. [BDF+20] provide empirical evidence of the concrete feasibility of both Paparazzi and replay attacks. Pietrzak et al. [ACK+21b] analyze inverse-sibyl attacks in which multiple adversaries cooperate to use the same pseudonyms. If one of the attackers gets to upload his TEKs, many false alerts may be raised. This attack could be used in combination with either the replay attack or our smart-contract based attacks in order to increase the number of affected targets. Iovino et al. [IVV21] concretely demonstrate the possibility to inject false alerts by replaying released TEKs. In particular, pseudonyms associated with already published TEKs are transmitted to smartphones whose clock is corrupted in order to make them believe these pseudonyms are valid for risk matching. They also show that several apps may publish TEKs that are still valid. These TEKs can be used to generate false alerts without the need of corrupting smartphones' clocks.

Several GAEN-based systems are currently used in the world for digital contact tracing. Vaudenay and Vuagnoux, and later Dehaye and Reardon, extensively evaluated SwissCovid [VV20,DR20b,DR20a], confirming some vulnerabilities showed in previous works and elucidating new ones. Finally, another class of attacks leading to false alerts involves bribing. Vaudenay envisions various possibilities for the development of dark economies [Vau20b] which could support false alert injection attacks, allowing them to be carried out at very large scales. In particular, the *Lazy Student* attack is connected to replay attacks. It is based on a dark economy where a hunter (i.e., seller) collects pseudonyms of individuals who will likely become infected later on, and deposits them on a smart contract. If the TEKs corresponding to such pseudonyms are uploaded to the server of the contact tracing system, the hunter gets a reward paid by a buyer (i.e., the lazy student). If replay attacks are doable, the buyer can use them to make target victims' apps raise false alerts. This dark economy is sustainable only if the smart contract has a way to check that pseudonyms were actually reported to the official server. Another form of dark economy described by Vaudenay is the *terrorist* attack. It involves users reporting pseudonyms that differ from the ones used during the previous days. In fact, in both Immuni and SwissCovid there is no mechanism forcing users to upload genuine TEKs. Again, a TEK could be posted on a smart contract automatically issuing a reward to whoever reports it to the contact tracing system. This purchase may lead to a massive amount of fake notifications, without relying on replay attacks.

On the (missing) risk assessment of the terrorist attack. The huge impact of false injection attacks seems to have gone unnoticed or just ignored. In [LHML20] the cybersecurity risks of contact

[8] In GAEN, depending on the particular application, this time may amount to up 14 days if the adversary colludes with the authorities, and to one day assuming TEKs are properly mixed and anonymized prior to publication.

tracing systems are reviewed and compared using a subjective scoring scheme. The report considers injection of false alerts notifications by only mentioning replay attacks or trivial attacks such as recruiting people with symptoms, while the terrorist attack is not even mentioned.

Vaudenay and Vuagnoux expressed these and other concerns in their analysis of SwissCovid [VV20]. The Swiss National Cyber Security Center (NCSC) answered to their criticism seemingly downplaying those risks. The possible development of dark economies was ignored [Swi20b] and a recap table on security issues reports on SwissCovid marks the concerns expressed by Vaudenay as addressed, including false alert injection attacks (see page 8 [Swi20c]) Nevertheless, no solution or mitigation to such problems is reported.

Bribing attacks on smart contracts. As we discuss in Section 7.2, our smart contracts make possible to trade TEKs reducing at a minimum the risks related to interacting with a dangerous entity such as a criminal. Bribery attacks on smart contracts for different scenarios have been proposed in the context of bribing miners in Ethereum and Bitcoin [MHM18, LK17, TJS16, VTL17, NKW21].

7.2 Trading TEKs in GAEN Systems

The GAEN API has been created to provide an efficient platform for exposure notifications on top of which countries can easily develop digital contact tracing systems. GAEN is supposed to solve various technical problems (e.g., changing BLE MAC address synchronously with the rotation of pseudonyms, keeping BLE advertisements on in background) on a large fraction of available smartphones[9]. At the same time, the API is so inflexible that it forces anyone who is willing to benefit from it to adopt a specific design for contact tracing. What is left in the hand of the developers is merely the creation of the graphical interface, the choice of some parameters and the realization of a server to gather and spread data about infected users and, more importantly, an authentication mechanism to avoid the upload of data by non-infected users.

Whenever a user is tested positive, she is given the right to upload her TEKs to the server so that the other users can be notified a risk of infection. The mechanism can be implemented in different ways. For example, a simple method consists of a code generated by the app that is given first to the health operator in order to activate it on the server. Then, once the server has authorized the code, the app will upload the TEKs along with the code (e.g., Immuni follows this approach). More complex mechanisms may be put in place. However, the attack we show next works for every GAEN-based contact tracing system under some natural assumptions that we will discuss later.

In order to evaluate the contagion risk, GAEN provides appropriate methods that take as input two files containing the last TEKs and the related signature. The matching is not performed if the signature does not verify under a public key previously known to Google and Apple. The first file is named `export.bin` and contains, along with other fields, a list of TEKs belonging to infected users that have decided to perform the upload procedure. Each TEK has also a date attached, which indicates when such TEK was used. The second file, named `export.sig`, contains a digital signature of the file `export.bin` [Goo20a, App20a]. An example of `export.bin` is reported in Sec. 7.2.3.

7.2.1 Take-TEK Smart Contract: Buying/Selling TEK Uploads

To simplify the description, we will refer to the TEKs file published by the server as a list of pairs of values. In each pair, the first value is a TEK and the second value is the corresponding date of usage date. Let the seller $\mathcal{P}$ be an infected user who would like to monetize her right to upload TEKs, and buyer $\mathcal{B}$ someone who is interested in paying $\mathcal{P}$ in order to upload TEKs of his interest. If the seller

[9]Indeed, see the case of UK that tried to develop a system without GAEN but had to give up https://www.bbc.com/news/technology-53095336.

can prove she acted as promised, this selling process can be executed remotely remaining automated, anonymous, and scalable. GAEN's design requiring the list of TEKs to be signed makes the verification easy to the smart contract, and greatly facilitates such trades. The trade can be performed using a blockchain capable of executing sufficiently powerful smart contracts (e.g., Ethereum). Such smart contract guarantees that $\mathcal{P}$ gets an economic compensation if and only if $\mathcal{P}$ uploads the TEKs specified by $\mathcal{B}$ to the server.

The high-level functioning of the smart contract is as follows. (1) $\mathcal{B}$ creates the smart-contract posting a list of TEKs with the related date, and deposits a prize to be redeemed by a seller. (2) An interested $\mathcal{P}$ also makes a small deposit to declare her intention to upload the TEKs specified by $\mathcal{B}$ (the purpose of this small deposit is explained later). After having made this deposit, (3) $\mathcal{P}$ has a specified amount of time to complete the upload procedure. Before the time runs out, $\mathcal{P}$ must provide a list of TEKs which includes all the pairs $(\mathsf{tek}, \mathsf{date})$ specified by $\mathcal{B}$, along with a valid signature under the server's public key. If $\mathcal{P}$ manages to do so, she gets a reward, otherwise both deposits go back to $\mathcal{B}$.

By making a deposit, the seller reserves a time slot during which she can perform the upload. Such deposit protects the buyer from denial of service (DoS) attacks by sellers who actually do not have the right to upload TEKs. Here, as in the remainder of the paper, with the word DoS we mean attacks carried out by fake sellers which prevent honest sellers from participating to the trade.

We name the above smart contract $\mathsf{Take\text{-}TEK}$ and the attack that leverages the use of this smart contract $\mathsf{Take\text{-}TEK}$ attack. The time window given to $\mathcal{P}$ must be wide enough to take into account that new TEKs are not continuously released by the server, in fact, several hours may pass between the submission of a TEK and its publication. Obviously, the amounts of both deposits will be significantly higher than transaction fees. A custom software is needed to upload arbitrary TEKs. However, this simple software may be developed even by other entities (not just the buyers), and publicly distributed on the Internet or other sources (e.g., Darknet). Therefore, all the seller would need to do is just running a software on a smartphone/computer, something that is easily doable by a large fraction of the infected citizens willing to gain money[10]. Additionally, the time given to the seller to complete the upload after having been tested positive must be long enough to reserve a slot on the blockchain (i.e., enough to wait that the transaction related to the seller's deposit gets confirmed) and subsequently send the TEKs via the custom software.

Attack description. $\mathcal{B}$ and $\mathcal{P}$ owns wallets $\mathsf{pk}_{\mathcal{B}}$ and $\mathsf{pk}_{\mathcal{P}}$ respectively. The buyer has no assurance that the seller is actually an infected person, and she is not just a malicious party trying to slow down the buyer's plan. Thus, some collateral must be deposited by $\mathcal{P}$ too. The seller will lose the collateral deposit in case she is not able to prove that she sent the buyer's TEKs to the server S. We use a signature scheme $(\mathsf{Gen}_{\mathsf{S}}, \mathsf{Sign}_{\mathsf{S}}, \mathsf{Ver}_{\mathsf{S}})$. The protocol description is depicted in Fig. 7.1 and a brief overview of the main functions follows below.

`Constructor`$(\mathbf{T}_{\mathcal{B}}, \mathsf{vk}_{\mathsf{S}}, t, d_{\mathcal{P}})$**:** It takes as input a set of tuples $\mathbf{T}_{\mathcal{B}} := (\mathsf{tek}_i^{\mathcal{B}}, \mathsf{date}_i^{\mathcal{B}})_{i\in[n]}$ with $n \leq \mathsf{maxteks}$ [11], where tek_i is the i-th TEK of the buyer and date_i is the associated date, the verification key vk_{S} to be used to verify the signature of the TEKs list, a timestamp t, indicating the maximum time the seller has to provide the correct list and signature, and the collateral value $d_{\mathcal{P}}$ that the seller must deposit.

`Deposit`()**:** must be triggered by $\mathcal{B}$ and takes as input a quantity p of coins as the payment for the seller.

`Promise`()**:** must be triggered by the seller $\mathcal{P}$ by sending a quantity of collateral deposit $d_{\mathcal{P}}$ as a payment when invoked.

[10] COVID-19 by itself caused a global economic crisis which led to lower wages and job losses. More details at https://en.wikipedia.org/wiki/COVID-19_recession.

[11] The maximum number of TEKs that can be uploaded in one shot depends on the particular contact tracing system.

`SendTeks`($\mathbf{T}_{\mathsf{KS}}, \sigma_T$)**:** must be triggered by the seller $\mathcal{P}$ to provide a list of TEKs together with its signature σ_T. Let the list released by the server be $\mathbf{T} = (\mathsf{tek}_i, \mathsf{date}_i)_{i\in[N]}$, where N is the number of published TEKs. It checks that:

- $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$ and $\mathsf{Ver}_{\mathsf{S}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathsf{S}}) = 1$.

If the checks pass, $d_{\mathcal{B}}$ coins are transferred to the seller's wallet $\mathsf{pk}_{\mathcal{P}}$.

Take-TEK Attack

We consider two entities: the seller $\mathcal{P}$ and the buyer $\mathcal{B}$, with wallets $\mathsf{pk}_{\mathcal{B}}$ and $\mathsf{pk}_{\mathcal{P}}$ respectively. The protocol works as follows:

1. $\mathcal{B}$ invokes the constructor, taking as input the buyer's TEKs list $\mathbf{T}_{\mathcal{B}}$, the server verification key vk_{S} that will be used to verify the signed TEKs list, a timestamp t, and a value $d_{\mathcal{P}}$ indicating the minimal amount that $\mathcal{P}$ must deposit in order to participate.
   After having created the contract, $\mathcal{B}$ triggers the function `Deposit` to deposit the prize p aimed for the seller who uploads $\mathbf{T}_{\mathcal{B}}$ to the server.
2. $\mathcal{P}$ deposits her collateral by triggering the function `Promise`. Now the seller has at most time t to send a TEKs list $\mathbf{T}$ signed by the server.
3. If $\mathcal{P}$, before time t, triggers the function `SendTeks` submitting a signed TEKs list $\mathbf{T}$ such that it satisfies conditions $\mathbf{T}_{\mathcal{B}} \subseteq \mathbf{T}$ and $\mathsf{Ver}_{\mathsf{S}}(\mathbf{T}, \sigma_{\mathbf{T}}; \mathsf{vk}_{\mathsf{S}}) = 1$, the collateral deposit $d_{\mathcal{P}}$ of $\mathcal{P}$ and the prize p are transferred to $\mathcal{P}$'s wallet. Otherwise, if t seconds have passed, they are moved to $\mathcal{B}$'s wallet.

Figure 7.1: The steps followed by buyer $\mathcal{B}$ and seller $\mathcal{P}$ to carry out the Take-TEK attack.

7.2.2 On the Practicality of Take-TEK Attack

Various proposed upload authorization mechanisms include manual steps (e.g., SwissCovid uses an authorization code, termed covidcode, which lasts for 24 hours) which, in order to function properly, naturally give the seller enough time to perform the steps mentioned in the section above. For example, if a code is communicated to the infected user via a phone call, she should be given a fairly large amount of time to write down the code and insert it in the app later on (the needs of people with disabilities and of elder people must be taken into account). Even systems that have fairly strict requirements on the time by which the upload procedure must be completed should allow for errors and recovery procedures, which may give additional time to the future seller. For example, Immuni requires that the infected user dictates, via phone call, a code that appears on her device. After that, the user must complete the upload within two minutes. If this does not happen, the procedure must be repeated. Additionally, the system should be tolerant. People should have the opportunity to perform the upload procedure later on if they are unable to do it in that precise moment. It is worth noting that strict requirements on the upload phase reduce users' privacy. A clear example is Immuni, where the medical operator, by checking whether a code has been used or is instead expired, gets to know whether or not the infected user actually uploaded her TEKs.

Implementation. We implemented our results as a smart contract for Ethereum, published it in a public repository[12] and tested it locally. Since Ethereum does not use `ECDSA-SHA256` (i.e., the one used in GAEN) for built-in transaction signature verification, there is the need to use specific solidity smart contract libraries[13] which lead to extra gas usage.

Considering the change of 206 dollars per single ETH token on the 20th of July 2020, signature verification costs around 11 dollars (1235000 of gas). In order to compute the full cost, one should add about 0.4 dollars (45000 of gas) for each TEK that is contained in the export.bin file[14]. Note that our smart contract can handle export files large as the maximum data that an Ethereum transaction can handle at most.

This limitation can be overcome by making the smart contract accepting the file split in multiple chunks (a transaction for each chunk), and then extracting the keys and verifying the signature by hashing the concatenation of all the stored chunks. A trivial solution to this problem can be to store $n-1$ chunks in the smart contract, and when the seller sends the n-th chunk, the smart contract performs the concatenation, extracts the keys, and verifies the signature. Unfortunately, storing data in a smart contract is the most expensive operation in terms of gas cost, and storing such a big piece of data in a smart contract state may be too expensive. However, exploiting the Merkle-Damgård construction used by SHA to hash multiple blocks, way less amount of data needs to be stored. Let us define `Hash` as the hashing algorithm and H_i as the hash of the i-th chunk C_i. TEKs extraction and signature verification in the chunk-based mechanism can be done in the following way:

- The seller divides the export file in different chunks in such a way that, when each chunk is hashed, the hash climbs up to the same level of the Merkle tree of the other hashed chunks.
- When the seller sends a new chunk to the smart contract, the latter extracts all the TEKs contained in the chunk, checks which of the buyer's TEK are present in the chunk and stores this information[15]. After that, it hashes the chunk and stores the hashed value H_i.
- When the last chunk is sent to the smart contract by the seller (together with the signature of the entire export file), the smart contract extracts the last pieces of information, checks if the TEKs contained in the last chunk cover the not yet appeared buyer's TEKs, computes its hash H_n, hashes its concatenation with the previously stored hashed chunks (i.e. it calculates $H_{\mathsf{out}} = \mathtt{Hash}(H_1, \ldots, H_n)$) and triggers the signature verification procedure giving the value H_{out} and the signature file as input.

As can be noticed, the application of the hashing algorithm to the concatenation of the H_is makes the hashing algorithm climbing up to the root of the Merkle tree, thus giving the expected hash of the entire file as output. Now the amount of bits needed to be stored is around $|H| \cdot n = 512 \cdot n$, vs $|C_i| \cdot n$ (usually the maximum transaction size, and so C_i in our case, is around 44 Kbytes in Ethereum).

7.2.3 Subtleties in the Wild

In 7.2.1 we gave a high-level overview of how TEKs uploads can be sold safely via blockchains. However, there are some subtleties we overlooked for the sake of simplicity. We first analyze the advantages for adversaries when using automated trade compared to already known attacks. Then, we consider certain problems that arise while trying to concretely mount our attack against deployed GAEN-based contact

[12] Code available at https://github.com/danielefriolo/TEnK-U.

[13] The one we used for signature verification is available at https://github.com/tdrerup/elliptic-curve-solidity.

[14] The cost of 45000 of gas includes TEK extraction, hashing of the export file for signature verification, checking if the stored TEKs are in the extracted ones. To simplify the gas evaluation, we assume that $\mathcal{B}$ stores only one TEK in the contract.

[15] During the chunk splitting, some TEKs may be cut in half. The smart contract should take care of the first and the last bits of each chunk and reconstruct the missing information.

tracing systems. We also show how these difficulties are easily tackled if very small modifications to our attack are made.

Advantages of automated trade (for an adversary). One might think that malicious injection of fake TEKs is inherent in decentralized contact tracing systems since there is no control over the smartphone used by infected individuals and thus, when the time of the upload comes, the infected person can always use a smartphone belonging to someone else.

While it is true that such simple attacks are very hard to tackle, they have limited impact for at least two main reasons: 1) the buyer must handover his smartphone to the seller, and this requires physical proximity; 2) sellers and buyers must trust each other since an illegal payment must be performed without being able to rely on justice in case of missing payment or aborted upload of keys. Indeed, even if in need of money, people are generally afraid of dealing with criminals since they may get scammed or threatened. Additionally, the buyer might expose the sellers' identities to the authorities in case he gets arrested or legally persecuted. Equally, the buyer may share the same concern with respect to an unreliable seller. It goes without saying that some citizens are prone to violate the rules[16] when they believe that risks are low compared to the advantages.
As such, attacks involving the exchange of smartphones, or the usage of a malicious app uploading TEKs sent by a criminal contacted directly by the infected citizen, do not scale and their damage may be considered tolerable. Having a mechanism which allows this trade to happen remotely, in anonymity and ensuring no party is cheated, solves all the above problems for parties willing to abuse contact tracing systems. In fact, it provides a framework for large-scale black markets of TEKs. The seller would not feel threatened in any way and could easily earn money, on the other hand, the buyers would benefit from a larger set of users to be in business with, therefore succeeding in many possible attack scenarios. Other systems for black markets based on reputations could also be used, but they are clearly less appealing than the transparency and usability of mediator-free smart contracts.

A worry-free seller. The effectiveness of a digital contact tracing system is strictly related to various factors among which the percentage of active population using them. Appropriate measures should be taken to earn citizens' trust since it is the only way to guarantee broad adoption. With this in mind, the European Commission released a series of recommendations in relation to data protection stating the need of identifying solutions that are the least intrusive and comply with the principle of data minimization [Eur20]. A similar recommendation has been given by the Chaos Computer Club (CCC) [Cha20] , the Europe's largest ethical hackers association, which explicitly states that "data which is no longer needed must be deleted". Corona-Warn, the German contact-tracing system, declares to be fully compliant with CCC's guidelines [Cor20] Many other systems are inspired by similar principles. For example, the Italian system Immuni also declares that data is deleted when no longer needed[17] as well as the Swiss system SwissCovid which also specifies a retention period for the TEKs and the upload authorization codes [18]. In its recommendation to build a verification server authenticating the uploaded TEKs, Google states that identifiable information should not be associated with uploaded data[19]. The adoption of the above measures ensures that uploaded data do not link to, nor identify a particular individual. This is very important considering that GAEN systems are vulnerable to the Paparazzi attack[20] [Vau20a].

[16] The infected person also commits a violation by allowing the injection of fake TEKs.
[17] https://github.com/immuni-app/immuni-documentation.
[18] Corona-Warn-App Solution Architecture https://github.com/corona-warn-app/cwa-documentation/blob/master/solution_architecture.md.
[19] Google: Exposure Notification Verification Server https://developers.google.com/android/exposure-notifications/verification-system.
[20] In *Paparazzi* attack, through passive antennas one can link pseudonyms used by an infected user tracing him over the duration of a TEK or for more days if the TEKs are linked. Therefore leaving open the possibility to link such data to a

Evaluation of seller's risks. Considering the above data minimization principles, are the seller and the buyer at risk of being legally persecuted for a trade that may be deemed as illegal? The answer seems to be no. If data is handled as specified above, there would be no way to associate the seller to its uploaded TEKs at a later time. Data exchanged during the attack would also not directly compromise neither the buyer nor the seller[21].

However, there is a problem for a seller who really wants to minimize the chance of getting caught. In fact, since the TEKs proposed by the buyer are posted in clear on the blockchain, authorities may become aware of them and activate ad-hoc procedures monitoring the incriminated TEKs and exploiting the upload authorization process to identify the guilty seller. This, in fact, does not seem to directly contradict the data minimization principle when national security is at stake. If the server getting the TEKs upload monitors the requests (e.g., by storing connection logs) without colluding with the health authority, the seller could be easily incriminated after the TEKs have been detected in the smart contract by just looking at her IP stored together with such request. However, in this case, the usage of an anonymity service like Tor [DMS04] can easily reduce the chance of getting caught. If the authorities are colluding, the upload authorization codes (e.g., the covidcode) may be associated with the identities of infected users, and TEKs could be in turn mapped to a precise individual via such codes. However, by slightly increasing the complexity of the smart contract, such risk may be completely avoided. It suffices for the buyer to encrypt his TEKs with a public key provided by the seller, who then will use a non-interactive zero-knowledge (NIZK) proof system to prove that the TEKs encrypted under the specified public key are indeed contained in the list signed with the server's public key. This requires an additional interaction with the buyer, who has to publish the encrypted TEKs (see the next paragraph for more details). Once again, the seller is protected by a timer which assigns her all the deposits if the buyer does not reply. Efficient Ethereum implementations of NIZK proofs are known in literature, like NIZKs for Σ-protocols [Wil18] or zk-SNARKs [Sem20, ZoK20, ZkD20].

Even if the buyer decides to claim the authorship of the attack at a later point in time (e.g., as it usually happens for terrorist attacks) by opening the encrypted values on the blockchain to published TEKs, the seller would not be at risk if data was handled according to the principles of data economy and anonymity. Any evidence based on contact tracing data would be a clear indicator that those principles have been violated. This could result in a big disincentive in using the app, since citizens may think (probably rightfully) that data could also be abused for other reasons, perhaps for mass surveillance purposes. Finally, we want to point out that even if several researchers raised the concern about the possible birth of black markets [Vau20b, RGK20], we did not find any document related to any contact-tracing system, either issued by governments or national security agencies, which deeply evaluates these risks. To the best of our knowledge, no risk analysis ever mentions to monitor the dark web and blockchains looking for suspicious smart contracts. It goes by itself that if blockchains are not monitored, all extra measures taken in this paragraph to protect the seller are not necessary.

Adding seller's privacy. Using publicly posted TEKs is dangerous for the seller due to possible risks of incrimination. This could disincentivize the seller to utilize such smart contract mechanism. To guarantee seller's privacy, in all of our attacks we can enrich our playground by assuming the existence of a CPA-Secure PKE encryption scheme $(\mathsf{Gen}, \mathtt{Enc}, \mathsf{VrfyOpen})$ and a NIZK proof system. The proposed protocols can be modified as follows:

- When the buyer creates the smart contract, he waits that a seller $\mathcal{P}$ is elected before providing his TEKs. When $\mathcal{P}$ is elected, $\mathcal{B}$ posts his TEKs encrypted with $\mathcal{P}$'s public key $\mathsf{pk}_{\mathcal{P}}$, by triggering

person's real identity would be extremely incautious.

[21] In this analysis, we refer only to contact tracing system data and messages exchanged via the blockchain during the execution of the attack. We do not take into account border-line situations as, for example, the case where there is only a single infected individual. We also ignore additional information that may help investigators figuring out who the seller is, for example how the money are spent after the trade.

an algorithm $\texttt{SendBuyerTeks}(\mathbf{C}_{\mathcal{B}})$ where $\mathbf{C}_{\mathcal{B}} = (c_1, \ldots, c_n)$, with $c_i \leftarrow\!\!\$\ \texttt{Enc}(t_i)$ for each $t_i \in \mathbf{T}_{\mathcal{B}}$. TEKs are pairs $t_i = (\mathsf{tek}_i, \mathsf{date}_i)$.

- When the signed TEKs list is available, the seller triggers $\texttt{SendTeks}(\mathbf{T}, \sigma_T, \Pi, \tilde{\mathbf{T}})$, where $\mathbf{T} = (\tilde{t_1}, \ldots, \tilde{t_N})$ are the published TEKs, σ_T the corresponding signature, and $\Pi = (\pi_1, \ldots, \pi_n)$ is a sequence of proofs in which π_i is a NIZK proof that the prover knows $t_i \leftarrow \mathsf{VrfyOpen}(c_i; \mathsf{sk}_{\mathcal{P}})$ and that at least one element $\tilde{t}_j$ in a subset $\tilde{\mathbf{T}} \subseteq \mathbf{T}$ such that $|\tilde{\mathbf{T}}| > |\mathbf{T}_{\mathcal{B}}|$ is equal to t_i. The smart contract checks all the proofs, and if all of them verify, transfers the prize to the seller.

Now the only information that an external observer can deduce by looking at the proofs is that all the encrypted buyer's TEKs are indeed inside the list (or in a subset of them). Depending on how the date field is handled it may be also necessary to encrypt it and to prove a slightly more complicated statement. To be sure that an observer cannot pinpoint the buyer's TEKs precisely, it is sufficient that the proofs use as a statement a subset of the published TEKs that contains at least one more TEK w.r.t. the buyer's TEKs (proving on a subset and not on the entire list can be beneficial in terms of proof size and efficiency). The only harmful case is when the number of published keys matches with the number of the buyer's keys. We can argue that this condition happens quite rarely, considering that one more external key is sufficient to guarantee buyer's safety, and if GAEN recommendations are followed, a decent amount of keys should be present in the list.

Other subtleties. Let's take an example of an `export.bin` file for Immuni, the Italian contact tracing app is reported below. The meaning of the main fields is commented on the side. The `start_timestamp` and `end_timestamp` are expressed in UTC seconds, the `rolling_start_interval_number` is expressed in 10 minutes increments from UNIX epoch. The `export.sig` contains the digital signature of the `export.bin` file, along with the field `signature_infos`. The content description of the `export.bin` file follows.

```
start_timestamp: 1591254000 //start of the time window of included keys
end_timestamp: 1591268399 //end of the time window of included keys.
region: "222"  batch_num: 1 batch_size: 1
signature_infos {
verification_key_version: "v1" //version of used verification key
verification_key_id: "222"
signature_algorithm: "1.2.840.10045.4.3.2"
1: "it.ministerodellasalute.immuni"}
keys {
key_data: ".." //base64 encoded TEK
transmission_risk_level: 8
rolling_start_interval_number: 2651616 //date of usage of TEK
rolling_period: 144}...
```

The following two other subtleties with limited impact shall be considered for the actual realization of the attack.

- **Extracting Public Keys from Signatures:** Take-TEK (cfr., Section 7.2.1) requires that the server's public key is known to both the involved parties. This guarantees that the buyer is sure the reward is paid only to sellers who actually upload data to the contact tracing system, and that honest sellers are sure they will be able to satisfy the conditions to be paid, namely obtaining a valid digital signature for reward redemption. A Github issue asking for the public key of the Italian contact tracing app was opened on the 7th of June 2020 and it has still not been

addressed at the time of writing. SwissCovid Android app contains a configuration file specifying the production version of the bucket public key (the value `BUCKET_PUBLIC_KEY` can be found in https://github.com/DP-3T/dp3t-app-android-ch/blob/master/app/backend_certs.gradle) that is used to perform signature verification outside GAEN. Anyway, as we can notice with Immuni, this is not a requirement.

One might think that keeping the verification keys secret may prevent attacks as the one of Section 7.2.1. However, it turns out that it is actually not the case. In fact, since GAEN uses ECDSA, starting from a signature and the related message we can recover two candidate public keys, one of which will match the actual one with overwhelming probability. A practical example showing this procedure can be found in [Yan19].

Such message/signature pairs are generally made publicly available and are easily accessible by appropriately querying the server of the specific contact tracing system. Multiple pairs per day may be released. A comprehensive description on how to get this data has been provided by the Testing Apps for COVID-19 Tracing (TACT) project, along with scripts to automate the downloading process [LF20a]. We also practically performed the extraction procedure, successfully extracting the keys for both SwissCovid and Immuni.

- **Updates of Public Keys:** There is a subtle technical problem with the attack described in Section 7.2.1. The digital signature keys that the server uses may change over time. In fact, the `export.bin` file includes a field indicating a version for the verification key. This field follows a progressive numeration, that is, the first version is termed v1, the second one v2 and so on. This means that the server may change the verification key it uses, perhaps within a set of keys that have been pre-shared with Google and Apple. Therefore, it might happen that, after the seller makes the deposit and accepts to upload the buyer's TEK, the server, by coincidence, decides to use a new key which was never used before, thus producing a signature that is not verifiable under the public key posted on the smart contract.

  However, by making a slight modification to the smart contract, it is possible to handle also this unfortunate event. Having realized that she would be unable to redeem the reward, the seller might activate a special recovery condition. After this, the buyer will be able to collect both deposits if and only if he manages to provide a pair of export files which have an `end_timestamp` subsequent to the time of the recovery request and verify under the public key originally posted on the smart contract; otherwise, the deposits are returned to the original owners. Obviously, enough time should be given to the buyer to provide the export files, similarly to what happens to the seller after her deposit.

  This event is certainly very annoying for the seller and might play as disincentive to join the trade, but taking a look at real-world data one realizes that this is a relatively rare event. We considered several countries which are currently using a digital contact tracing system, namely: Italy, Switzerland, Austria, Germany, Ireland, Northern Ireland, Denmark, Latvia, Canada and US Virginia. Until January 13th 2021 (last time we checked), only US Virginia and Italy have switched to the second version of the verification key. In particular, the change to the Italian system dates back to the 15th of June 2020[22] and no modifications have been made since then. Notably, some countries' systems, like Switzerland and Germany's ones, are active from several months now and the verification key has not changed at all. To the best of our knowledge, the criteria by which the verification key should change is not documented anywhere.

[22] This change occurred in the 4th export file.

7.3 Connecting Smart Contracts to TLS Sessions

The Take-TEK attack relies on the fact that a digital signature is used to authorize uploads. Additionally, the ability to extract the public key from signed messages may also play a key role. Therefore, one might think that to protect GAEN systems, the public key should remain hidden and the signature scheme should be such that extracting the public key from message-signature pairs is hard. In this way, due to the inability of allowing a smart contract to verify that a TEK is officially in a list of infected TEKs, the attack would fail. However, things are not so easy. The previous smart contract exploited the public verifiability of the signatures because this is what is used in GAEN systems. If a different method is used, it might be abused again. Indeed, we show that TLS oracles can be used to prove to a smart contract that an upload was successfully performed, without relying on signatures of TEKs.

7.3.1 Decentralized Oracles

Recently, Zhang et al. [ZMM+20], introduced the concept of Decentralized Oracles. Roughly, an oracle is an entity that can be queried by a client to interact with a TLS server and help the client proving statements about the connection transcript. Previously known oracle constructions rely on trusted/semi-trusted execution environments [ZCC+16], thus not giving any help in our case. DECO [ZMM+20] is the first work where a fully-decentralized construction is proposed for specific ciphersuites such as CBC-HMAC and AES-GCM coupled with DH key exchange with ephemeral secrets. We recall that a TLS connection is divided in two parts: a handshake phase where key exchange is performed, and a phase during which the transferred data is encrypted/decrypted by the client/server using the key exchanged in the previous phase. GAEN servers usually accept Elliptic-Curve Diffie-Hellman key Exchange (ECDHE) for the first phase, while for the second phase some servers accept only AES-GCM (e.g., Immuni), whereas others, like SwissCovid's one, accept also CBC-HMAC as a ciphersuite. To guarantee the integrity of data, the plaintext is usually compressed and a MAC on the compressed string is calculated using a key derived from the DH exchanged key.

Decentralized Key-Exchange. We provide below an informal description of the key-exchange in DECO for ECDHE that is called Three Party Handshake (3PHS).

We assume three entities: a prover $\mathcal{P}$, a verifier $\mathcal{V}$ and a server S. $\mathcal{P}$ and $\mathcal{V}$ jointly act as a TLS client. The overall idea of DECO is that the prover and verifier, after performing some two-party computations, compute shares of the exchanged key, while the server computes the entire key without even noticing that $\mathcal{P}$ and $\mathcal{V}$ are two distinct interacting entities.

When using CBC-HMAC, the keys $k_{\mathcal{P}}^{\mathtt{MAC}}$, $k_{\mathcal{V}}^{\mathtt{MAC}}$ (such that $k_{\mathcal{P}}^{\mathtt{MAC}} + k_{\mathcal{V}}^{\mathtt{MAC}} = k^{\mathtt{MAC}}$) are learned by $\mathcal{P}$ and $\mathcal{V}$ respectively, while $k^{\mathtt{Enc}}$ is only known to $\mathcal{P}$. When using AES-GCM, the same key is used for both encryption and MAC, therefore both $\mathcal{P}$ and $\mathcal{V}$ just get a share of it. While $\mathcal{P}$ and $\mathcal{V}$ only learn their secret shares of the key, the server S gets to know both $k^{\mathtt{Enc}}$ and $k^{\mathtt{MAC}}$. Let G be an EC group generator. The key exchange phase works as follows:

- $\mathcal{P}$ establishes a TLS connection with the server S.
- When receiving the DH share $Y_{\mathsf{S}} = s_{\mathsf{S}} \cdot G$ from S, $\mathcal{P}$ forwards it to $\mathcal{V}$.
- $\mathcal{V}$ samples a DH secret $s_{\mathcal{V}}$ and sends his DH share $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ to $\mathcal{P}$.
- $\mathcal{P}$ samples her DH secret $s_{\mathcal{P}}$, calculates her DH share $Y_{\mathcal{P}} = s_{\mathcal{P}} \cdot G$, calculates the combined DH share $Y = Y_{\mathcal{P}} + Y_{\mathcal{V}}$, and sends Y to S.

Finally, S computes the DH exchanged key as $Z = s_{\mathsf{S}} \cdot Y$. $\mathcal{P}$ and $\mathcal{V}$ will compute their secret shares of Z as $Z_{\mathcal{P}} = s_{\mathcal{P}} \cdot Y_{\mathsf{S}}$ and $Z_{\mathcal{V}} = s_{\mathcal{V}} \cdot Y_{\mathsf{S}}$. Note that $Z_{\mathcal{P}} + Z_{\mathsf{S}} = Z$, where $+$ is the EC group operation. Now that $\mathcal{P}$ and $\mathcal{V}$ have secret shares of EC points, they use secure two-party computation (2PC) to

evaluate a PRF (that we call TLS-PRF) to derive the keys $k_{\mathcal{P}}^{\mathtt{MAC}}$ and $k_{\mathcal{V}}^{\mathtt{MAC}}$. The authors face and solve several challenges in order to derive keys efficiently via 2PC. We do not cover this part, a more detailed description can be found in [ZMM+20].

Encrypted communication. At the end of the 3PHS, $\mathcal{P}$ and $\mathcal{V}$ have to engage in a 2PC protocol to correctly calculate the MAC and the encryption on the plaintext to be sent to the server, without revealing the shares to each other. Privacy of the plaintext is also ensured with respect to $\mathcal{V}$. For CBC-HMAC, the encryption of such plaintext is computed exclusively by $\mathcal{P}$ who holds the encryption key. The authors [ZMM+20] provide hand-optimized protocols which are much more efficient than the ones obtained by directly applying 2PC techniques. The 2PC protocol for AES-GCM is a lot slower than the one for CBC-HMAC since for AES-GCM $\mathcal{P}$ and $\mathcal{V}$ must cooperate also for the encryption.

Proving statements. An important feature of DECO is that $\mathcal{P}$, when the communication with S ends, can prove, in zero knowledge, statements on the communication transcript in a clever and efficient way. However, to make their protocol practical for our goal, we do not try to maintain the transcript private. As a result, we will not discuss this part of DECO which can be found in [ZMM+20]. In the following, we describe how to adapt DECO to our scenario.

7.3.2 A Smart Contract Oracle

Our goal is to make the smart contract play the role of the DECO verifier. In this way, the smart contract would be able to verify that the intended communication between the seller and the server took place and to reward the seller accordingly. Unfortunately we can not just plug DECO into a smart contract for several reasons. For example, DECO requires to run intensive 2PC related tasks, to sample random values and to maintain a private state[23].

Therefore, we keep running the DECO protocol off-chain but we find a way to connect the DECO run between the prover and the verifier to the state of the smart contract, so that the smart contract will eventually be able to act as an impartial judge punishing the malicious party when a deviation from the prescribed honest behavior is detected. In particular, the seller acts as a prover and the buyer as a verifier, and we guarantee no party is able to cheat (i.e., the seller is paid if and only if she performs the upload of the requested TEKs) by binding the off-chain execution to the state of the smart contract itself. Furthermore, we guarantee privacy of the messages exchanged between the server and the prover only until their TLS connection is open. After the communication ends, the seller proves that she acted honestly by providing the application-level messages exchanged with the server, along with the corresponding MAC tags w.r.t. the MAC key which is bound to the smart contract. To be more specific, the smart contract freezes a share of the MAC key and the seller has to show a communication transcript (i.e., the messages exchanged with the server and corresponding MAC tags) which is consistent with such share. Privacy of the upload request message to be sent to the server is crucial while the TLS session is open because the verifier may abort the protocol and use the authorization token of the prover to upload data by himself without paying out the promised reward. On the other hand, making the communication public after it took place does not endanger the prover, apart from the considerations made in Section 7.2.3 , and makes the verification procedure much more practical. What we need is that the shares of the prover and the verifier are kept private until the end of the protocol, and then revealed to the smart contract, along with other information, for verification and reward paying. In addition, the TLS session timeout should be big enough to allow for the 2PC execution. To this regard, Zhang et. al already verified the practical feasibility of their protocol [ZMM+20]. Obviously, $\mathcal{P}$ must know how to reach $\mathcal{V}$ to carry out the protocol. To address concerns regarding anonymity, $\mathcal{V}$ may set

[23] Keeping a private state inside a smart contract is not possible and computationally intensive operations generate high costs.

up a Tor hidden service[24]. Using hidden services may significantly slow down the process, however,s we found both Immuni and SwissCovid servers to give a generous time out window of two hours[25]. Another point to consider is that upload authorization tokens may have a limited duration. For example, in SwissCovid, the smartphone, by interacting with an appropriate server (different from the TEKs upload server, called CovidCode-Service), exchanges the covid code for a signed JWT token that is valid for 5 minutes[26]. Then, this token is sent by the smartphone to the server along with the TEKs to complete the upload. Thus, the upload message, containing the TEKs and the authorization token, must be computed and sent to the server within 5 minutes from the reception of the JWT token. Given the high efficiency of DECO when CBC-HMAC is used, even when bandwidth is limited [MMZ+20], it is reasonable to think that the attack is feasible in SwissCovid. In Immuni instead, no signed token is used. In fact, the upload must be completed within 2 minutes after the infected user has communicated the code to the health operator. Therefore, in Immuni the attack would less likely be operative, especially with Tor, given that the slower AES-GCM ciphersuite is required.

Protocol description. From now on, we refer to the seller and the buyer as prover $\mathcal{P}$ and verifier $\mathcal{V}$ respectively; we denote the server as S. In the following, we explain the detailed attack for the CBC-HMAC ciphersuite. When creating the smart contract, $\mathcal{V}$ also posts the DH share $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ he is willing to use during the 3PHS, along with requested TEKs (and dates).

First, $\mathcal{P}$ transacts on the smart contract to reserve a time slot of duration t_1 by which a DECO protocol run must be performed together with $\mathcal{V}$ and S, and the data needed to redeem the reward must be posted on the smart contract by $\mathcal{P}$. If time t_1 elapses, $\mathcal{P}$ loses her slot. This reservation mechanism is needed to prevent $\mathcal{V}$ from getting back the reward while an honest $\mathcal{P}$ performs the upload of the requested TEKs. In fact, the verifier could also act as a prover and simulate a reward-paying interaction with the server to the smart contract, which would have no mean to distinguish it from a fake one. By adding a reservation mechanism, we are sure a malicious $\mathcal{V}$ cannot play a simulated transcript in the smart contract while honest $\mathcal{P}$ is performing with him the DECO protocol run. Furthermore, since the communication for the upload between the server and the prover consists of just a single query followed by a single response, it is not possible for a cheating verifier to make the timer expire avoiding to pay the prover while at the same time the upload of the TEKs successfully completes. In fact, once $\mathcal{V}$ cooperates with $\mathcal{P}$ to build a valid request, S will reply to $\mathcal{P}$ independently of what $\mathcal{V}$ does, thus giving $\mathcal{V}$ all she needs to redeem the reward.

When executing the 3PHS, $\mathcal{P}$ checks that the value $Y'_{\mathcal{V}}$ sent by $\mathcal{V}$ during the handshake corresponds to the value $Y_{\mathcal{V}}$ posted on the smart contract. This prevents $\mathcal{V}$ from providing an erroneous DH share and blaming $\mathcal{P}$ for it. If this is not the case, $\mathcal{P}$ aborts. Since no upload message has been sent to the server yet, no party gains advantage from this operation. If $\mathcal{V}$'s share is correct (i.e., $Y_{\mathcal{V}} = Y'_{\mathcal{V}}$), parties engage in the communication with S and jointly compute the MAC (via 2PC as in [ZMM+20]) on the upload request m_c generated by $\mathcal{P}$. If the connection ends successfully[27], the elected $\mathcal{P}$ posts (only who reserved this slot is allowed to post this message) on the smart contract the following:

- The entire communication transcript, that is (m_c, m_s) together with the MACs (θ_c, θ_s), calculated by the client(s) $\mathcal{P} \leftrightarrow \mathcal{V}$ and the server S.
- The prover's secret $s_{\mathcal{P}}$.

[24] More on Tor hidden services can be found at https://2019.www.torproject.org/docs/onion-services.

[25] Interestingly, in June the timeout of a TLS session with both Immuni and SwissCovid upload servers was limited to 5 minutes, but it has been then extended to two hours.

[26] See CovidCode-Service configuration https://github.com/admin-ch/CovidCode-Service/blob/develop/src/main/resources/application-prod.yml.

[27] This can be inferred from the communication. For example, as in SwissCovid (see SwissCovid Server Controller: https://github.com/DP-3T/dp3t-sdk-backend/blob/a730a5b276591e5cc8b6c609e2b0ba29c6069eb6/dpppt-backend-sdk/dpppt-backend-sdk-ws/src/main/java/org/dpppt/backend/sdk/ws/controller/GaenController.java), S may reply $\mathcal{P}$ with either a success message such as "200 OK" or an error message.

- The DH share of the server Y_{S} received during the 3PHS.

Then, the smart contract starts a timer t_2 indicating the maximum time $\mathcal{V}$ has to reveal his secret $s_{\mathcal{V}}$. In case $\mathcal{V}$ does not do that, the prize is automatically transferred to the seller $\mathcal{P}$. If $\mathcal{V}$ reveals $s_{\mathcal{V}}$, the smart contract does the following:

- Check that $Y_{\mathcal{V}} = s_{\mathcal{V}} \cdot G$ and if not, transfer the prize to $\mathcal{P}$.
- If the check passes, reconstruct the secret Z from $s_{\mathcal{V}}, s_{\mathcal{P}}, Y_s$, and apply TLS-PRF to derive the MAC key $k^{\mathtt{MAC}}$.

Now the smart contract has everything it needs to check that the fields inside message m_c (from the prover to the server) are correct (i.e., the buyer's TEK are present), the response message (from the server to the prover) is positive, and that the MACs (θ_c, θ_s) verify w.r.t. $k^{\mathtt{MAC}}$. If all the checks pass, the prize is transferred to $\mathcal{P}$, otherwise $\mathcal{P}$ gains no prize and the deposit is returned back to $\mathcal{V}$.

As mentioned before, $\mathcal{V}$ is not encouraged to provide a different public key w.r.t. the one he used in DECO execution, otherwise $\mathcal{P}$ will just abort. On the other hand, the prover is not able to earn a reward without uploading the promised TEKs. In fact, the probability for the prover to come up with a pair (m'_c, θ'_c) (resp. (m'_s, θ'_s)) that verifies under the key $k'^{\mathtt{MAC}}$ derived from $Z' = Z'_{\mathcal{P}} + Z'_{\mathcal{V}}$ with $Z'_{\mathcal{P}} := s'_{\mathcal{P}} \cdot Y'_{\mathsf{S}}$ and $Z_{\mathcal{V}} := s_{\mathcal{V}} \cdot Y'_{\mathsf{S}}$ is negligible due to the fact that $s_{\mathcal{P}}$ is fixed and honestly generated, thus randomizing Z', hence $k'^{\mathtt{MAC}}$.

A note on DoS attacks. It is important to prevent DoS attacks run by sellers who actually do not have the right to upload TEKs and end up by just wasting the buyer's precious time. In the previous discussion this protection is not provided: before sending the jointly computed message (m_c, θ_c), the seller can decide to not forward the message to the server. Now, the buyer has to open his commitment to show his secret $s_{\mathcal{V}}$ in order to not lose the prize. As a result, the committed value cannot be used in other runs. To address this issue, the smart contract can be modified to handle multiple sessions. Instead of storing $Y_{\mathcal{V}}$ as a single DH contribute, the buyer stores the root of a Merkle tree. Now, when the seller interacts with the contract to reserve a session, a session id (a simple counter j suffices) is assigned to her: the DH contribute used in the 3HPS will correspond now to the j-th leaf of the Merkle tree. Now, when the buyer has to open his secret $s_{\mathcal{V}}$, he also reveals the path of the Merkle tree from the root to the leaf j. The smart contract will now verify that the contribute is correctly derived from the root by following a path with correct openings. Let us consider a Merkle root committing to 2^k elements, thus allowing the buyer to open as many sessions. For a k large enough, a malicious seller should spend a considerable amount of money in order to reserve all the sessions.

AES-GCM. Carrying out the attack when AES-GCM ciphersuite is required is more involved. Differently from CBC-HMAC, AES-GCM relies on the same key for both encryption and MACs. The impact of AES-GCM is twofold: 1) more computation is needed to perform the required 2PC to calculate messages from/to the server, due to the AES algorithm itself, 2) the prover does not learn the encryption key after 3PHS, meaning that both encryption and decryption must be done via 2PC as well. On the smart contract side, this difference boils down to a lack of fairness. After $\mathcal{V}$ and $\mathcal{P}$ have calculated together the upload message and sent it then to S, $\mathcal{V}$ could decide not to help the prover to decrypt the server's response. Now, $\mathcal{P}$ has no witness in her hands to give to the smart contract in order to prove that she has correctly performed the TEKs upload. As a result, she cannot redeem the prize. The problem can be easily solved by giving to the smart contract the burden of decrypting the server's ciphertext. In our approach, $\mathcal{V}$ must commit to his key and open it later. When this happens, the server reconstructs the MAC/encryption key, decrypts the ciphertext, does the necessary checks, and pay the prize to $\mathcal{P}$. The CBC-HMAC version of DECO is way faster then the AES-GCM one. However, looking at practical evaluations made by the authors [ZMM+20, MMZ+20] it is reasonable to

think that all their solutions may fit in the time window given by contact tracing servers (e.g., 2 hours in Immuni and SwissCovid) for the TLS connection, even when hiding $\mathcal{V}$ through Tor hidden services. What is less likely is that, in the case of Immuni which uses AES-GCM and requires the upload to be completed within two minutes, the upload request message (m_c, θ_c) is computed and sent to the server in time; especially when the prover and the verifier communicate via Tor.

7.4 Conclusion

We showed that the terrorist attack, previously envisioned by Vaudenay, is concretely realizable against GAEN systems with the aid of cryptographic tools and a blockchain capable of executing smart contracts (e.g., Ethereum). In particular, the Take-TEK attack exploits the fact that the list of infected TEKs, published by the server daily, has always a digital signature attached to it. Such a signature allows the smart contract to easily verify that the upload was performed as requested by the terrorist. Even beyond the use of signatures, we have shown a different instantiation of the terrorist attack using DECO. In conclusion, we advise protocol designers not to look at the effects of a specific realization, but to prove the protocol secure against any automated instantiation of a terrorist attack. Our work shows that the power of blockchain technology to trade digital assets is still overlooked even when critical features, like contact tracing of infected individuals, are digitized.

Chapter 8

Practical Verifiable MPC using Bulletproofs/AC20

8.1 Introduction

Electronic voting protocols are prime examples of secure multi-party computation (MPC) that separate input and compute parties. In this setting, there are many input parties that outsource the tallying operation to a set of compute parties. This setting introduces new requirements versus classical MPC protocols where parties are considered both input and compute party.[1] A necessary requirement for voting protocols is public verifiability [CF85, BY86, Ben87, CGS97]. While voting protocols specialize in the linear operation of tallying votes, the focus of this chapter is a scheme that defines *publicly verifiable MPC for general arithmetic circuits*.

We present a practical scheme in which the task of the input party is reduced to a minimum: The input party posts a single encrypted message to a bulletin board. The compute parties then apply a threshold cryptosystem to transform the encryption to secret shares, to be used as input for the secure computation. The compute parties also produce a zero-knowledge proof of correctness of the computation that allows anyone, particularly someone external to the secure computation, to check the correctness of the output, while preserving the privacy properties of the MPC protocol.

The structure of our scheme has the following similarities with the voting scheme of Cramer *et al.* [CGS97]. The scheme separates active parties into input parties and computation parties. To achieve public verifiability all parties have access to a bulletin board, which can receive encryptions from the input parties. Due to the homomorphic properties of the encryption method, the computation is publicly verifiable. The use of a matching fault-tolerant threshold decryption technique ensures that the inputs remain private and failure of compute parties can be tolerated up to a given threshold.

Our scheme addresses the challenge of general arithmetic circuits using recent results in zero-knowledge proof systems, particularly compressed Σ-protocols [AC20] and Bulletproofs [BBB+18a]. This chapter introduces a practical construction based on [AC20], which reconciles Bulletproofs with Σ-protocol theory. The construction yields proofs of logarithmic size and does not require a trusted setup, i.e., the setup does not require knowledge of a trapdoor.

8.1.1 Active Security versus Public Verifiability

Goldreich et al. [GMW87] made the observation that any MPC protocol that is secure against passive adversaries can be made secure against active adversaries by requiring all parties to prove correctness for each message sent. However, correctness cannot be guaranteed in general. For example, if interactive zero-knowledge (ZK) proofs are used, an adversary controlling all parties is able to falsify proofs.

[1] Think of Yao's Millionaires' Problem [Yao82] for example.

Cramer et al. [CDN01] introduced a new approach to MPC basing it on homomorphic threshold cryptosystems, secure against an active adversary that corrupts any minority of compute parties. Building on [CDN01], [Hoo12] introduced the notion of universally verifiable secure computation by requiring that all parties send *non-interactive* zero-knowledge proofs. We will refer to the notion of universally verifiable MPC as *publicly verifiable* MPC, or *verifiable* MPC for short.

Breakthroughs in non-interactive proof systems led to the first practical constructions for verifiable MPC. Particularly, the Pinocchio proof system [GGPR13, PHGR13] introduced a practical zk-SNARK that allowed a prover to perform cryptographically verifiable computations with verification effort less than performing the computation itself. Building on Pinocchio, Schoenmakers *et al.* [SVdV16] introduced the publicly verifiable MPC protocol Trinocchio [SVdV16]. The zk-SNARK required a trusted setup that was specific to the computation, however.

8.1.2 Properties of the Practical Construction

Following [Ben87, CGS97], we require the verifiable MPC construction to be practical, publicly verifiable with reusable and non-trusted setup, and private.

Practical means that for input length n and circuit length m, the secure computation has communication complexity $O(\log(m + n))$ and computation complexity $O(m + n)$, expressed in secure multiplications.

Publicly verifiable means that anyone with access to the bulletin board can verify the correctness of the computation. To further enhance practicality versus prior constructions, we require that the cryptographic setup for the proof system is reusable for circuits up to a given size and that the setup does not need to be performed by a trusted party.

Private means that no action can be taken by any minority of active parties which will leak information about the inputs other than what can be inferred from the result of the computation.

8.1.3 LIBOR as a Motivating Example

To illustrate the usefulness of verifiable MPC, we describe its application to setting the LIBOR benchmark interest rates without a trusted party.

The London Interbank Offered Rate (LIBOR) is a set of benchmarks that reflect the average interest rate at which large global banks can borrow from each other. It is an important metric to price loans and is produced once a day by the Intercontinental Exchange (ICE) asking a panel of contributing banks "At what rate could you borrow funds [..] just prior to 11 a.m. London time?" The banks then confidentially send their answers to ICE, who calculates the mean of the inputs in the second and third quartile.

The Libor scandal arose when it was discovered that banks were falsely inflating or deflating their rates to profit from trades, or to give the impression that they were more creditworthy than they were.[2]

Our construction could replace ICE as a trusted party, while ensuring confidentiality for the contributing banks and public verifiability. Particularly, public verifiability is achieved by asking the auditor of each participating bank to verify, cryptographically sign and then publish the bank's encrypted input to the daily computation. By verifying the signatures on the encrypted inputs and the cryptographic proof of computation, the general public receives strong guarantees that the secure computation is correctly performed on verified inputs, i.e. that no tampering happened on the input or the computation.

[2] See https://en.wikipedia.org/wiki/Libor_scandal.

8.1.4 Our contribution

The main result of this chapter is a conceptual verifiable MPC scheme and practical construction that takes as inputs encryptions on a bulletin board, has a reusable and non-trusted setup based on standard cryptographic assumptions, and permits encrypted outputs.

The conceptual scheme requires a bulletin board, a public key cryptosystem and a non-interactive proof system that can refer to encryptions from this cryptosystem in its statement.

Our practical construction uses the [AC20] proof system based on Bulletproofs [BBB+18a], requiring intractability of the discrete logarithm problem and the Diffie-Hellman problem. The compressed Σ-protocol for circuit satisfiability [AC20, Protocol 8] refers to Pedersen vector commitments of the circuit's inputs in its statements. We use the homomorphic properties of ElGamal to implement a threshold cryptosystem that ensures the inputs remain private and to refer the proof of correct computation to the encrypted inputs on the bulletin board.

We present the first implementation of [AC20] as a Python library. To simplify compilation of arithmetic circuits required for the proof system, the Python library automatically constructs an arithmetic circuit for a given snippet of Python code. Consistent with the design philosophy of MPyC [Sch18] it uses operator overloading to construct a datastructure that represents the arithmetic circuit with addition, scalar multiplication and regular multiplication gates.[3]

Our Python library exploits the property that for many functions, verification can be done more efficiently once the result has been computed, possibly requiring a little extra computation. This resembles the fact that an NP-statement can be verified in polynomial time given a witness, whereas finding such a witness need not be possible in polynomial time. The circuit compiler will replace the computational circuit by a more efficient verification circuit for this purpose. I.e., during evaluation of an operator at runtime, the compiler records the arithmetic gates and inputs that correspond to the verification of the application of that operator.

8.2 Building Blocks

Our scheme combines several primitives, namely a bulletin board, a threshold cryptosystem and a verifiable MPC protocol, using a non-interactive proof system. The bulletin board hosts the (encrypted) inputs and outputs of the computation, including cryptographic proofs. The threshold cryptosystem enables encrypted inputs and outputs to the verifiable MPC protocol. Supporting our threshold cryptosystem is the notion of *secure groups*, a scheme to implement finite groups as oblivious data structures presented in [pri20, pri21]. Throughout, let p be a prime, $\mathbb{Z}_p$ be shorthand for $\mathbb{Z}/p\mathbb{Z}$, $\mathbb{F}_p$ denote a field of prime order p, and boldface $\boldsymbol{a} \in \mathbb{F}_p^n$ denote an input vector of length n.

8.2.1 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]. 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 $a \in \mathbb{F}_q$.

In the context of exponentiation and integer division, $[\![a]\!]_{\mathbb{Z}}$ 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. When it is directly clear from the context, we typically suppress $_{\mathbb{Z}}$ or the modulus in the notation of a secret share.

[3] Each gate has fan-in two and unbounded fan-out.

8.2.2 Bulletin Board

The communication model required for our scheme is a public broadcast channel with memory, which we will refer to as a *bulletin board*. All communication through the bulletin board is public and can be read by any party (including passive observers). No party can erase any information from the bulletin board, but each active participant can append messages to its own designated section. For a recent example of a security framework and construction of a bulletin board protocol, we refer to [KKL+18].

8.2.3 Secure Groups

For a finite group $\mathbb{G}$, a secure groups scheme defines the secure representation of group elements and secure implementation of group operations, including doubling, exponentiation and inversion, random sampling and encoding to/from secure group elements.

Definition 6 ([pri20, pri21]). *Let $\mathbb{G}$ be a finite group. A* **secure group scheme** *comprises protocols for the following tasks, where $a, b \in \mathbb{G}$.*

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 $[\![\sigma^{-1}(a)]\!]$.*

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.

To construct *secure group elements*, one could proceed as follows. Let a map ϱ permit a representation of a group element $a \in \mathbb{G}$ to a tuple or matrix of finite field elements. Given $a \in \mathbb{G}$, define $[\![a]\!]_{\mathbb{G}}$ as the coordinate-wise application of $[\![\ ]\!]$ to $\varrho(a)$.

8.2.4 Threshold Cryptosystem

We define the following threshold cryptosystem: Let $\mathbb{G}$ be a cyclic group with generator g of large prime order p. Given protocols for secure groups, extend a classical $(t+1, M)$-threshold ElGamal cryptosystem [Ped91] with the following protocols:

Encryption of shared message. Given message $[\![a]\!]_{\mathbb{G}}$, the parties generate $[\![u]\!]$ with $u \in_R \mathbb{Z}_p$, and output ciphertext for public key h as the pair $(A, B) \leftarrow (g^{[\![u]\!]}, h^{[\![u]\!]}[\![a]\!]_{\mathbb{G}})$.

Threshold decryption to shared message. Given ciphertext (A, B), compute $[\![A^x]\!]_{\mathbb{G}} = A^{[\![x]\!]}$. The parties compute and keep message $[\![a]\!]_{\mathbb{G}} = [\![B]\!]_{\mathbb{G}} / [\![A^x]\!]_{\mathbb{G}}$ in shares.

8.2.5 Circuit Satisfiability Proof System

A verifiable MPC protocol requires a zero-knowledge proof system for the circuit satisfiability problem, i.e., for a given arithmetic circuit C a prover shows that it knows an input $\boldsymbol{a} \in \mathbb{F}_p^n$ for which $C(\boldsymbol{a})$ evaluates to 0.[4] In a verifiable MPC protocol, MPC parties compute a non-interactive zero-knowledge proof for the circuit satisfiability relation:

$$R_{cs} = \{(C; \boldsymbol{a}) \colon C(\boldsymbol{a}) = 0\}. \tag{8.1}$$

The proof system consists of the following protocols:

Setup. Given a cyclic group $\mathbb{G}$ of prime order, compute a set G of generators such that it is hard for provers (compute parties) to find nontrivial linear relations between the generators in G. In our construction $|G|$ is dependent on and linear in the number of input gates, n, and multiplication gates, m, of the largest circuit to be used.[5]

Prove. Given secret-shared input $[\![\boldsymbol{a}]\!]$, encryption $Enc(\boldsymbol{a})$ and circuit C, compute zero-knowledge proof π for relation $\{(C, Enc(\boldsymbol{a}); \boldsymbol{a}) \colon C(\boldsymbol{a}) = 0\}$.

Verify. Given proof π and statement $(C, Enc(\boldsymbol{a}))$, verify in zero-knowledge the correctness of $C([\![\boldsymbol{a}]\!]) = 0$.

Generally, a proof system requires a cryptographic setup, which can be trusted or untrusted and support specific or general circuits. For simplicity, the setup in the above scheme corresponds to the setup step of [AC20], which is reusable for circuits up to a given size and does not require a trusted party with knowledge of a trapdoor. The setup for the Pinocchio zk-SNARK [PHGR13] would require a trusted party to generate computation-specific evaluation and verification keys using secret random elements.

8.3 Practical Construction

8.3.1 Verifiable MPC using the AC20 Proof System

[AC20] reconciles Bulletproofs [BBB+18a] with Σ-Protocol theory. The main construction of [AC20], based on the discrete log assumption, has communication complexity and proof size that are both logarithmic in the circuit size. We outline the components of this proof system and recall the main theorem underpinning the circuit satisfiability protocol.

Components of the AC20 Proof System.

The protocol combines two essential components. The first component is a Σ-protocol that yields zero-knowledge proofs for arbitrary linear statements and uses an adaptation of the compression technique of Bulletproofs to reduce communication complexity. This protocol is referred to as compressed Σ-protocol Π_c in [AC20, Protocol 5].

Second, to enable ZK proofs of non-linear statements, the authors apply an adaptation of [CDP12] to construct a Σ-protocol proving the correctness of one compact commitment to m multiplication triples $(\alpha_i, \beta_i, \gamma_i \leftarrow \alpha_i\beta_i)$, corresponding to the left input, right input and output wires of the circuit's multiplication gates, with low amortized complexity for large m. This technique is presented in circuit satisfiability protocol Π_{cs}, [AC20, Protocol 8].

[4] Note that this directly generalizes to relations where $C'(a) = b$, by defining $C(a, b) \leftarrow C'(a) - b$.

[5] For simplicity, we assume a setup as in [AC20]. A setup for Pinocchio [PHGR13] would require a trusted party to generate evaluation and verification keys based on secret random elements.

Finally, the complete Σ-protocol for circuit satisfiability, [AC20, Protocol 8], combines the above two components and performs an amortized nullity check on the three linear forms corresponding to the multiplication gates' left input, right input and output wires, plus the linear forms corresponding to the circuit's outputs.

Σ-protocol for Circuit Satisfiability.

We briefly elaborate on these last two steps. Given a circuit C with m multiplication gates, define α_i, β_i, γ_i, for $i \in \{1, \ldots, m\}$, as above, i.e., the values of the left input, right input and output wires of the circuit's mul-gates. Define u_i, v_i as the affine forms in input vector $(\boldsymbol{a}, \gamma_1, \ldots, \gamma_m)$ that correspond to the α_i, β_i for $i \in \{1, \ldots, m\}$ and given input $\boldsymbol{a}$. Let $\boldsymbol{\alpha}$ denote $(\alpha_1, \ldots, \alpha_m)$, and define $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ simillarly.

The Σ-protocol for showing correctness of the multiplication triples follows these steps[6]:

- Select two random polynomials $f(X), g(X) \in \mathbb{Z}_p[X]_{\leq m}$ that define a packed secret sharing of the vectors $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ respectively. Compute $h(X) \leftarrow f(X)g(X)$.
- The prover commits to $\boldsymbol{z} = (\boldsymbol{a}, f(0), g(0), h(0), ..., h(2m)) \in \mathbb{Z}_p^{n+2m+3}$ and sends it to the verifier.
- The verifier selects a random challenge $c \in \mathbb{Z}_p \backslash \{1, \ldots, m\}$ and sends it to the prover.
- The prover sends $y_1 \leftarrow f(c)$, $y_2 \leftarrow g(c)$, $y_3 \leftarrow h(c)$ to the verifier who checks $y_3 = y_1 y_2$.

The final step of Π_{cs} is to perform an amortized nullity check on the three affine forms corresponding to all multiplication gates' left input, right input and output wires, plus the set of affine forms corresponding to the circuit's outputs. The amortized nullity check uses the compressed Σ-protocol Π_c as a black box for opening linear forms and then applies a polynomial amortization technique to do many nullity checks at once, on the committed vector $\boldsymbol{a}$, with low amortized complexity (see [AC20, Protocol 7]).

Using Lagrange interpolation, the affine forms in the coefficients of $\boldsymbol{z}$ corresponding to the multiplication triples evaluated in c, denoted by f_c, g_c and h_c, are constructed as follows:

$$f_c \leftarrow \sum_{j=0}^{m} u_j(\boldsymbol{a}, f(0), \gamma_1, \ldots, \gamma_m) \prod_{i \neq j} \frac{c-i}{j-i} \tag{8.2}$$

$$g_c \leftarrow \sum_{j=0}^{m} v_j(\boldsymbol{a}, g(0), \gamma_1, \ldots, \gamma_m) \prod_{i \neq j} \frac{c-i}{j-i} \tag{8.3}$$

$$h_c \leftarrow \sum_{j=0}^{2m} h(j) \prod_{i \neq j} \frac{c-i}{j-i}. \tag{8.4}$$

Given input $\boldsymbol{a}$ and the values of the output wires of the multiplication gates, $\boldsymbol{\gamma}$, the circuit's output gates can also be directly expressed as affine forms in coefficients of $\boldsymbol{z}$. Our circuit compiler, using Python's operator overloading feature, directly performs the construction of these required forms for Π_{cs}.

Theorem 5 ([AC20]). *Π_{cs} is a $(2\mu+5)$-move protocol for relation R_{cs}. It is perfectly complete, special honest-verifier zero-knowledge and computationally knowledge sound, under the discrete log assumption, with knowledge error*

$$\kappa \leq \frac{2\mu + 2m + 5}{p - m}, \tag{8.5}$$

where $\mu = \lceil \log_2(n + 2m + 4) \rceil - 1$ and given one compact commitment to a vector of inputs in $\mathbb{Z}_p$.

[6] We refer to [AC20, Section 6.1] for the validity of this protocol.

Performing Secure Group Operations in MPC.

Let n be the maximum number of inputs and m be the the maximum number of multiplication gates of the circuits used. Given a finite cyclic group $\mathbb{G}$ of prime order p and computation inputs in $\mathbb{F}_p$, constructing a verifiable MPC protocol from Π_{cs} requires the following *secure group* operations from Definition 6:

- Creating a set of $n+2m+3$ independent generators G of $\mathbb{G}$ can be implemented using a function to *encode* a pseudo-random number to a group element.
- Computing the Pedersen vector commitments and announcements (elements A and B in compressed Σ-protocol Π_c) requires *secure exponentiation* of a public group element by a secret-shared exponent.

Final Remarks on the Circuit Satisfiability Protocol.

The Fiat-Shamir heuristic is applied to make the circuit satisfiability protocol Π_{cs} and its sub-protocols, most importantly $(2\mu+3)$-move Π_c, non-interactive: Every next challenge is computed as a hash of the statement, together with all previous messages of the protocol. Unfortunately, for a $(2\mu+3)$-move public coin protocol the security loss of the reduction equals $Q^{\mu+1}$, where Q is the number of random oracle queries by the adversary. In [ACK21a, Section 5.5] hypothesize that the actual security loss is in the order of Q.

For input length n and circuit length m, the secure computation has communication complexity $O(\log(m+n))$ and computation complexity $O(m+n)$, expressed in secure multiplications.

A program with constant-sized proofs based on the Knowledge-of-Exponent assumption is also presented in [AC20, Section 9]. To implement Π_{cs} for constant-sized proofs our implementation also supports finite groups based on the pairing-friendly BN256 curve with homogeneous projective coordinates.

8.3.2 ElGamal Ciphertexts as Inputs

We will use ElGamal ciphertexts to provide secret inputs to a secure computation, focusing on the basic case that the input value is a bit $b \in \{0,1\}$. Concretely, we consider ElGamal ciphertexts of the form $(g^u, h^u g^b)$, where the private key $x = \log_g h$ is shared by the MPC parties. The ciphertexts may be posted to a bulletin board.

The MPC parties apply threshold decryption to convert the encrypted bit b into a secret-shared bit $[\![b]\!]$, using the following threshold Elgamal cryptosystem. 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 compute g^x. The parties keep private key $[\![x]\!]$ in shares and output public key $h = g^x$.

Encryption. Given message $b \in \{0,1\}$, pick $u \in_R \mathbb{Z}_p$. The ciphertext for public key h is the pair $(g^u, h^u g^b)$.

Threshold decryption. Given ciphertext (A, B), the parties use $[\![x]\!]$ and the secure group exponentiation protocol with secret output to compute $[\![A^x]\!]_{\mathbb{G}}$. The parties then compute $[\![g^b]\!]_{\mathbb{G}} = B/[\![A^x]\!]_{\mathbb{G}}$ and perform a secure decoding to obtain $[\![b]\!]$.

The MPC parties also need to compute a Pedersen commitment $P = h_2^r g^b$, and prove consistency with the given ciphertext (A, B). This amounts to providing a Σ-proof for the following relation:

$$\{(A, B, P; x, b, r) : B = A^x g^b,\ P = h_2^r g^b\} \tag{8.6}$$

Extension to vector commitments:

$$\{(\boldsymbol{A}, \boldsymbol{B}, P; x, \boldsymbol{b}, r) : \forall_{i=1}^{n} B_i = A_i^x g^{b_i},\ P = h_2^r \prod_{i=1}^{n} g_i^{b_i}\} \tag{8.7}$$

Efficient Σ-protocols for these relations are easily obtained by standard techniques.

The MPC parties then perform the secure computation of $C(\boldsymbol{b})$, along with the computation of a compressed Σ-proof for relation $\{(C, P; \boldsymbol{b}) : P = h_2^r \boldsymbol{g}^{\boldsymbol{b}}, C(\boldsymbol{b}) = 0\}$.

8.3.3 One-Time Pads as Outputs

We provide a trivial solution to enable secret outputs for the verifiable MPC scheme: MPC parties first compute $C([\![\boldsymbol{a}]\!]) = [\![b]\!]$ and then add a random one-time pad $[\![r]\!]$ to the output. The circuit satisfiability proof is now applied to secret input vector $(\boldsymbol{a}, b, r)$ and slightly adapted circuit C' defined as $C'(\boldsymbol{x}, y, z) \leftarrow C(\boldsymbol{x}) - (y + z)$, which should evaluate to 0 for input vector $(\boldsymbol{a}, b, r)$.

8.4 Software

The verifiable MPC Python package supporting the construction of Section 8.3 is available from https://github.com/toonsegers/verifiable_mpc under the MIT license. It requires the secure groups extension, which is available from https://github.com/toonsegers/sec_groups under the MIT license. The verifiable MPC package includes a convenient circuit compiler as well as several so-called gadgets to construct verification circuits for operators (problem statements) that are more efficient than the associated computation circuits.

The Python package extends the MPyC framework [Sch18] to enable multiple MPC parties to express a verifiable MPC computation in Python code. The package implements the [AC20] proof system, particularly the circuit satisfiability protocol Π_{cs}. It offers the option to use the compressed pivot protocol Π_c, which is based on the discrete log assumption, and yields proofs that are of size $O(\log(m+n))$. The second option is to use the program based on the Knowledge-of-Exponent assumption, yielding constant sized proofs. This second option includes a KEA-based vector commitment scheme together with a ZK protocol for opening of linear forms. This ZK protocol replaces the compressed Σ-(sub)protocol Π_c in Π_{cs}.

8.4.1 Abstraction for Secure Group Operations

To expedite the development of verifiable MPC we designed and implemented an extensive secure group scheme. The secure group scheme lets us operate easily and efficiently on secret-shared group elements. In principle, any finite group can be made secure this way. We have included simple examples like small permutation groups, but we mostly focus on groups that are used for cryptographic purposes such as elliptic curves.

The secure groups Python package[7] simplifies the implementation of the Π_{cs} protocol and directly yields two scenarios in one go: single prover and multiple MPC parties acting as one prover. The overhead is very modest, both in terms of cognitive load for the engineer and computational efficiency versus a direct implementation of Π_{cs} in Python. Implementing Π_{cs} via the secure group abstraction, on top of the MPyC framework [Sch18], abstracts away most of the implementation details of necessary MPC sub-protocols in the setting for information theoretically secure MPC, tolerating a dishonest minority of passively corrupt parties.

[7] Available from https://github.com/toonsegers/sec_groups under the MIT license.

8.4.2 Circuit Compiler

To simplify compilation of arithmetic circuits required for the proof system, we include the option to automatically construct an arithmetic circuit for a given snippet of Python code. Consistent with the design philosophy of MPyC [Sch18], it uses operator overloading to construct a data structure that represents the arithmetic circuit with addition, scalar multiplication and regular multiplication gates.

8.4.3 Gadgets

Our Python library exploits the property that for many functions, verification can be done more efficiently once the result has been computed, possibly requiring a little extra computation. This resembles the fact that an NP-statement can be verified in polynomial time given a witness, whereas finding such a witness need not be possible in polynomial time. The circuit compiler will replace the computational circuit by a more efficient verification circuit for this purpose. That is, during evaluation of an operator at runtime, the compiler records the arithmetic gates and inputs that correspond to the verification of the application of that operator.

Some problems for which verification is more efficient than solving the problem:

- $b = \sqrt[k]{a}$ can be verified by $a^k = b$;
- $b = a^{-1}$ can be verified by $ab = 1$;
- Integer division $(q, r) = (\lfloor a/b \rfloor, a \mod b)$ can be verified by $a = bq + r$ and $0 \leq r < b$;
- Extended gcd (g, α, β) of a and b can be verified by $g = a\alpha + b\beta$, $g|a$, $g|b$, and $g > 0$;
- Bit decomposition $(a_0, \ldots, a_{\ell-1})$ of a can be verified by $\forall_{i=0}^{\ell-1} a_i \in \{0, 1\}$ and $a = \sum_{i=0}^{\ell-1} 2^i a_i$;
- Zero-test $b = (a \neq 0)\ ?\ 1 : 0$ can be verified by calculating witness $c = (a + (1 - b))^{-1}$ and checking that $ac = b$ and $a(1 - b) = 0$.

Gadgets can be directly implemented in the CircuitVar class of the library by specifying the verification equations and inputs for the given operator.

8.5 Conclusion

This chapter presents a conceptual scheme and practical construction for verifiable MPC, a cryptographic scheme that can be viewed as a natural extension of electronic voting to general arithmetic circuits. The practical construction is based on standard cryptographic assumptions and has a transparent setup. A proof-of-concept implementation in Python is included with this deliverable (see previous section).

The main ingredient for verifiability is the generation of zero-knowledge proofs in MPC. As our implementation can also be run with a single party, it constitutes a true generalization of zero-knowledge proofs. The generalization comprises the replacement of a single prover by a set of $m \geq 1$ provers where at most $t < m/2$ provers may be corrupt.

Acknowledgements. We would like to thank Thomas Attema for his valuable suggestions.

Chapter 9

Verifiable Multi-Party Business Processes

Business process automation (BPA) is the use of technology to execute recurring tasks or processes with the goal of replacing manual effort. Most BPA deployments aim to automate a firm's internal operations. However, many business processes are composed of a series of steps taken by different firms. Single-party business processes such as invoice production and processing, which might be handled using a corporate enterprise resource planning (ERP) system, can be viewed as steps in the overarching multi-party business process. Thus, if BPA aims to improve performance of a firm's internal business activities, we may define multi-party business process automation (MPBPA) as the use of technology to optimize and automate firms' interactions with each other.

The challenges in realizing MPBPA differ from those solved by existing BPA technology. The most important difference is the inherent lack of trust between firms on matters of value. A firm would be naturally hesitant to allow a computer program to automatically pay invoices; instead, review and approval by a trusted employee in accounts payable is typically required. If it were possible to automate such interactions in a trusted way, many of the same benefits that BPA has yielded within the scope of individual firms could be realized at an inter-company or even systemic level. The existence of current manual reconciliation processes shows that firms are willing to pay a substantial premium to ensure that business rules, designed to protect against fraud or error, are applied correctly.

Distributed ledger technology (DLT) can be described as the process of replicating and synchronizing shared data between several mutually distrusting parties. Byzantine fault tolerant (BFT) consensus protocols are typically used for the replication and synchronization, and are often seen as the underlying technological basis for DLT. These consensus algorithms, however, present a number of challenges. First, BFT consensus has high network overhead. This has caused many practical implementations to appoint a small subset of nodes in the network as designated validators who receive all transactions. The validators then apply the business rules, which are encoded in the ledger itself as so-called smart contracts, to determine which transactions are valid, and follow the consensus protocol to agree on how the ledger state should be updated.

Second, clients might not want to share their confidential business data. In such cases, only the parties to a contract would receive the transaction details and know the code to be executed. Thus, the parties execute the code independently and only post attestations about the current state of the contract to the ledger. The validators follow the consensus protocol to ensure that the attestations are reliably recorded. In this case, the validators do not know the transaction data or the contract code nor whether the contract has been executed correctly. The attestation of state is still useful, however, since all parties can be assured of having a common, non-repudiable view of the state. We refer to this arrangement as using "privacy limited validation."

We propose an approach that is more scalable and can be used as the basis for many of the multiparty

process enforcement use cases that others address through DLT. Our solution is based on committing process states into an authenticated data structure (ADS) operated by a server whose actions can be independently verified.

9.1 Related work

Several DLT-based business process management (BPM) systems have been proposed to support execution of multi-party processes.

Caterpillar [LPGBD+19] is a BPM system that runs on top of the Ethereum blockchain. It keeps all process related data on the blockchain, in order to ensure its reliability. On one hand, no off-chain data is required to read or execute a process. On the other hand, potentially sensitive data must be posted to the blockchain. Lorikeet [TLW18] is a model-driven engineering tool. It generates smart contract code (in the Solidity language of the Ethereum blockchain) from a Business Process Model and Notation (BPMN) representation. It uses DLT for communication between parties, but sensitive data is not posted to the blockchain. There are other solutions outside of academic research, such as the Proof of Process by Stratumn.[1] All of these suffer from the inherent limitations of DLT design, namely the overhead required to reach consensus between parties. This limits the throughput of these systems. A comparative overview of these systems is given in [DCCD+19].

The Laava platform [WLT+19] focuses on multi-tenant aspects of BPM and proposes to solve these by creating a private blockchain instance for each tenant and periodically aggregating the states of these private chains into a commitment posted to a public blockchain as an external anchor. A drawback of this approach is that a user would still have to scan a whole private blockchain to extract, prove, or verify the history of one process.

Mendling et al. [MWA+18] discuss different directions in business processes and blockchain interaction. According to their taxonomy, our approach is based on monitoring. It does not put restrictions on execution of business processes, but provides a secure, performant, and scalable way to monitor the execution of multi-party processes. Breu et al. [BDE+13] introduced the distinction between process orchestration, where the process instance is managed by a central service provider, and process choreography, where the ownership of the instance is transferred from one participant to another. We note that our monitoring-based approach is applicable in both cases and perhaps even more valuable in the context of choreographies where it is easier for the parties to get confused over the current state of the process [PSHW20].

9.2 Our approach

Verifiable Business Processes (VBP) is a transaction-based server-centric solution for adding trust to a registry or a service in order to achieve verifiable multi-party business process automation. In a distributed ledger, new transactions are approved by majority agreement; VBP instead is based on single trust domain where it generates proofs of correct operation, which would make incorrect behavior immediately evident.

We use an authenticated map of key-value pairs maintained in a sparse Merkle tree (SMT) [DPP16]. A VBP server maintains one such tree and populates it incrementally, starting from an empty tree. For better performance, the server operates in rounds, collecting updates within a time period and committing them as a batch at the end of the round. All modifications are public for the participants of the process, which enables them to audit the operation of the server if they wish to do so. The root of the SMT is the trust anchor for the proof holder. It is critical for a verifier to know that only one root value is produced at the end of each round. This guarantees there exists only one version of the

[1] https://www.proofofprocess.org

tree for all participants. Our solution uses a set of external auditors who digitally sign the roots of the SMT for that purpose.

The SMT, by itself, cannot be used to describe a business process. We assume that the allowed states and transitions of each process are defined in some formalism. This could be based on a general standard, such as BPMN, or some custom formalism. VBP tracks the state of each process instance as a key-value pair where the key is a permanent and unique process identifier and the value represents the current state of the process instance. The hash of the process identifier determines the leaf of the SMT for tracking the evolution of the state of that process instance and the hash of the current state is stored in the designated leaf of the SMT.

As a process is executed, participants send hashes of the updated process states to the VBP server. The server records each update in the corresponding leaf of the SMT. This enables extraction of a proof of state for that process instance relative to the root hash of the SMT. When a process instance is altered to a new state, the hash value for this process will change. Anyone might see that this has occurred, but without having access to the underlying process instance data that is hashed, it reveals nothing. Thus, the VBP server functions as an independent witness of the process state, but without having to know anything about the transactions or the business rules, in the manner described as the "privacy limited validation" above.

How the process instance moves from one party to another is beyond the scope of VBP. This could be done by any means of transport agreed upon by the participants. In all cases, the integrity of the process instance is guaranteed by the VBP server. Note that the VBP server does not receive any sensitive data to provide that service. Only meta-data on when processes are started and modified is required.

Our current implementation uses a JavaScript based process definition library to describe the process verification rules. Process definitions are modeled as finite state machines (FSM). The same library generates representations of process instances. These are not based on BPMN, although it would be possible to generate these artifacts from a BPMN tool.

The evolution of process instances can be modeled as a new SMT being constructed by the VBP server for each round. To ensure that all participants have a consistent view of the evolution of a process, we need to prove that the server is not maintaining parallel histories for a process instance and showing different histories to different participants.

Eijdenberg et al. discuss the properties of verifiable log backed maps (VLBM) where the history of changes to a simple SMT is recorded in a verifiable log for auditing purposes.[2] However, full audit is required to prove correct operation of a VLBM. Since a full audit would be impractical in large trees, it could allow a malicious operator to perform flip-flop attacks with low probability of detection.

The data structure used in VBP improves upon the VLBM in this respect. Instead of just the hash of the state of a process instance, each leaf of the SMT in VBP stores a record (H, C, T) where H is the hash of the current state, C is the counter of state changes since the process instance was created, and T is the time of the last state change (expressed as the number of the round when that change was recorded in the SMT).

This data structure enables efficient auditing of the VBP server in a manner similar to the server auditing process described in [BLT18]. For efficiency, the VBP server applies updates to the SMT in batches. An auditor keeps the current value of the root hash of the SMT as its internal state and uses that to verify the correctness of the updates submitted by the server. The correctness of the batch of updates that transforms the SMT with the root hash R in round N to the SMT with the root hash R' in round $N + 1$ is checked as follows:

1. The auditor sets R^* to R, to start from the SMT state as of round N.

2. For each process state update, the server presents

[2] https://www.continusec.com/static/VerifiableDataStructures.pdf

- the record $X = (H, C, T)$ of the process instance state as of round N;
- the hash chain A linking X to R^*;
- the updated record $X' = (H', C', T')$.

3. The auditor processes each of those updates by
   - checking that A indeed links X to R^*; this establishes the correctness of the initial state of the record;
   - checking that $C' = C + 1$ and $T' = N + 1$; this establishes the correctness of the update of the metadata in the record; note that the auditor does not check the state change, because it is auditing the behavior of the VBP server and not of the business process participants;
   - updating R^* to the output value of the hash chain A when the record X is replaced with X'; note that the hash chain A is the same as in the first check, which ensures that the server could not have changed any other records with this update.
4. After processing all updates, the auditor checks that $R^* = R'$; this ensures that the server has presented exactly the set of updates that transformed the SMT from the state with the root R to the state with the root R'.
5. If all the checks pass, the auditor signs the new root hash value R' as approved and the server can use it as a reference relative to which proofs can be delivered to clients and also as the starting state for the next batch of updates.

Crucially, the audit protocol allows multiple auditors to work in parallel and independently sign their approvals, thus avoiding the need for a distributed consensus protocol, which would reintroduce some of the limitations of DLT.

For clients, the VBP server provides an interface for querying the (H, C, T) records, together with the associated hash chain proofs. This enables efficient traversal of records relating to a particular process. The client can query the latest state of the process, then use the time T of the last change to query the previous state at time $T - 1$, and so on until the history of the process instance has been traced back to its creation. The client can then verify that all the state transitions are indeed valid according to the agreed process model. The client can also verify that the counter of state changes increases by one with each change, which ensures that there are no other changes to the process state outside this reported history.

This mode of operation prevents the possibility that processes could be altered briefly, in collusion with the VBP server operator, for the purpose of producing a fake "proof" of an incorrect state, after which the process would be put back to the correct, unaltered state. This type of fraud mainly affects parties who are not participating in the process, but have some outside interest in it, and rely upon its accuracy for some reason. Without a mechanism to prevent it proactively, this type of fraud can only be detected by full audit.

9.3 Performance

Keeping the SMT with full modification history takes storage space. Each internal node of the SMT contains a hash value of k bits. In the leaf nodes, the size of the hash value dominates over the two integers, so for simplicity we can assume equal-sized nodes. As only the hashes of the process states are kept in the SMT, the space requirement depends only on the number of process instances and their modification rates, not on the size of process states.

When a full SMT would be naively stored for each round, each process instance would fill one leaf node and cause at most k internal nodes to be populated with non-empty values, for a total of about

$N \cdot M \cdot k$ nodes for M processes maintained over N rounds. However, if only a minority of the processes change in each round, most nodes of the SMT remain the same from one round to the next and those nodes can be shared between the two trees, thus reducing the storage to $N \cdot C \cdot k$, where C is the average number of process instances that change per round. Since the tree is a sparse one, most paths from a non-empty leaf to the root have many empty siblings. Neither these siblings nor the parents computed from them need to be stored, as each such parent can be re-computed on demand from the only non-empty child. With this optimization, only $\log_2 M$ nodes need to be kept on an average path, for a total of $N \cdot C \cdot \log_2 M$ nodes, each containing k bits, or $k/8$ bytes of data.

The process count is not a constant, however. The append-only nature of the SMT means it is always growing. We can assume constant rate of new process instances to get some useful estimates. For example, with 1 000 clients and 1 000 new process instances per client per year, and an average of 10 state modifications per process instance, we get the rate of about 10 million updates per year and the tree with 1 million leaves by the end of the first year, 2 million leaves by the end of the second year, etc. It is up to the application to decide how long historical state should be stored. A rolling window approach (e.g., store only last year's worth of full proof history) could be applicable in many cases.

9.4 Conclusions and Outlook

We have proposed an alternative to distributed ledgers for adding integrity to multi-party business process automation. This solution uses a "trust, but verify" model, which allows us to create a highly scalable solution. In addition, our solution does not require each participant to install new infrastructure. The central piece—the VBP server—could be shared between many participants in a software-as-a-service (SaaS) model.

Our main contribution is adding efficient auditability of modification history to the sparse Merkle tree (SMT) based authenticated data structure, which protects clients against malicious behavior by the service provider, including advanced threats, such as the "flip-flop" attack (cf. Sec. 9.2). The other contribution is showing how such an authenticated data structure could be used in multi-party business process automation use cases.

A limitation of VBP is that it can handle only linear processes. Extending the model to support branching, parallel execution, and merging is an interesting avenue of future research.

Currently, a previous version of VBP is used in one production system for monitoring the business process execution and easing the auditing process of the Certus service.[3] However, the solution is not yet integrated into any established BPM systems. This is a possible direction of future practical development effort. As the VBP server is agnostic to what data it contains, multiple such integrations could be supported even by a single instance of the VBP service.

[3] https://www.certusdoc.com

Chapter 10

Conclusions

This document contains new protocols and results for distributed ledgers. In the deliverable, we have presented the revisions made to the core protocols presented in D3.2 and new results based on the feedback and experience with the WP1 use cases.

We have also included results that despite not being strictly related to Use Cases and the Toolkits are of great interest for the distributed ledger technology landscape. As such, we recall that most of the results presented in the document have already been published in well-known security and cryptography conferences.

In the deliverable we have discussed how to securely update existing blockchain protocols, how to enhance the privacy of a blockchain system without introducing additional trusted assumptions, proposed a new Byzantine fault-tolerant (BFT) total order broadcast protocol aimed at maximizing throughput on wide-area networks, and discussed how to use the blockchain to improve the efficiency and the verifiability aspects of a specific class of multi-party computation protocols.

We also considered the important problem of understanding to what extent is possible to remove data from blockchains like Bitcoin, and we focused the remaining part of the document on discussing how the distributed ledger technology could play a role in the problem of contact tracing.

Bibliography

[a8x] a8x9. a8x9. https://github.com/a8x9.

[a8x20a] a8x9. DP-3T. https://github.com/DP-3T/documents/issues/66, 2020.

[a8x20b] a8x9. DP-3T. https://github.com/DP-3T/documents/issues/210, 2020.

[AAC+08] Yair Amir, Yair Amir, Brian Coan, Jonathan Kirsch, and John" Lane. Byzantine replication under attack. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*, 2008.

[ABB+18a] 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 Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15, 2018.

[ABB+18b] 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 Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018.

[AC20] Thomas Attema and Ronald Cramer. Compressed Σ-protocol theory and practical application to plug & play secure algorithmics. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 513–543, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[ACK21a] Thomas Attema, Ronald Cramer, and Lisa Kohl. A compressed Σ-protocol theory for lattices. *IACR Cryptol. ePrint Arch.*, 2021:307, 2021.

[ACK+21b] Benedikt Auerbach, Suvradip Chakraborty, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak., Michael Walter, and Michelle Yeo. Inverse-sybil attacks in automated contact tracing. In *Proc. of CT-RSA*, volume To appear, 2021.

[AFV21] Gennaro Avitabile, Daniele Friolo, and Ivan Visconti. Tenk-u: Terrorist attacks for fake exposure notifications in contact tracing systems. In *Proceedings of 19th International Conference on Applied Cryptography and Network Security '21*, volume To appear, 2021.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[AGM+17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017.

[AKWW19] Georgia Avarikioti, Lukas Käppeli, Yuyi Wang, and Roger Wattenhofer. Bitcoin security under temporary dishonest majority. In Ian Goldberg and Tyler Moore, editors, *FC 2019: 23rd International Conference on Financial Cryptography and Data Security*, volume 11598 of *Lecture Notes in Computer Science*, pages 466–483, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019. Springer, Heidelberg, Germany.

[AMQ13] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: redundant Byzantine fault tolerance. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 297–306, 2013.

[AMVA17] Giuseppe Ateniese, Bernardo Magri, Daniele Venturi, and Ewerton Andrade. Redactable blockchain–or–rewriting history in bitcoin and friends. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 111–126. IEEE, 2017.

[App20a] Apple. Setting up an Exposure Notification server. https://developer.apple.com/documentation/exposurenotification/setting_up_an_exposure_notification_server, 2020. Accessed: 2020-08-23.

[App20b] Apple and Google. Apple and Google's exposure notification system. *https://www.apple.com/covid19/contacttracing*, 2020.

[Bai20] Leemon Baird. Personal communication, 2020.

[BAL20] Francesco Buccafurri, Vincenzo De Angelis, and Cecilia Labrini. A privacy-preserving solution for proximity tracing avoiding identifier exchanging. *CoRR*, abs/2005.10309, 2020.

[BBB+18a] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[BBB+18b] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[BCG15] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. https://eprint.iacr.org/2015/1015.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 31–60, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

[BDE+13] R. Breu, S. Dustdar, J. Eder, C. Huemer, G. Kappel, J. Köpke, P. Langer, J. Mangler, J. Mendling, G. Neumann, S. Rinderle-Ma, S. Schulte, S. Sobernig, and B. Weber. Towards living inter-organizational processes. In *2013 IEEE 15th Conference on Business Informatics*, pages 363–366. IEEE, 2013.

[BDF+20] Lars Baumgärtner, Alexandra Dmitrienko, Bernd Freisleben, Jonas Höchst, Mira Mezini, Markus Miettinen, Thien Duc Nguyen, Alvar Penning, Filipp Roos, Ahmad-Reza Sadeghi, Michael Schwarz, and Christian Uhl. Mind the gap: Security & privacy risks of contact tracing apps. In *TrustCom 2020, Security Track*, September 2020.

[BDFG20] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. Cryptology ePrint Archive, Report 2020/1536, 2020. https://eprint.iacr.org/2020/1536.

[BDK15] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pages 173–184, 2015.

[Ben87] Josh Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, September 1987.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 677–706, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[BGG19] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *FC 2018 Workshops*, volume 10958 of *Lecture Notes in Computer Science*, pages 64–77, Nieuwpoort, Curaçao, March 2, 2019. Springer, Heidelberg, Germany.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[Bit] Bitcoin. http://bitcoin.org.

[Bit19] Bitcoin visuals: Transaction sizes. https://bitcoinvisuals.com/chain-tx-size, 2019.

[BLS19] Johannes K Becker, David Li, and David Starobinski. Tracking anonymized bluetooth devices. *Proceedings on Privacy Enhancing Technologies*, 2019(3):50–65, 2019.

[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.

[BMS19] Stefano Bistarelli, Ivan Mercanti, and Francesco Santini. An analysis of non-standard transactions. *Frontiers in Blockchain*, 2:7, 2019.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[BNPS03] Mihir Bellare, Chanathip Namprempre, David Pointcheval, and Michael Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *J. Cryptol.*, 16(3):185–215, 2003.

[BSA14] Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *International Conference on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32:824–840, October 1985.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. M.Sc. Thesis, University of Guelph, Canada, June 2016.

[But] Vitalik Buterin. On-chain scaling to potentially 500 tx/sec through mass tx validation. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477.

[BY86] Josh Cohen Benaloh and Moti Yung. Distributing the power of a government to enhance the privacy of voters (extended abstract). In Joseph Y. Halpern, editor, *5th ACM Symposium Annual on Principles of Distributed Computing*, pages 52–62, Calgary, Alberta, Canada, August 11–13, 1986. Association for Computing Machinery.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[Cap18] Daily hodl: Cryptocurrency transaction speeds: The complete review. https://dailyhodl.com/2018/04/27/cryptocurrency-transaction-speeds-the-complete-review/, 2018.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

[CDP12] Ronald Cramer, Ivan Damgård, and Valerio Pastro. On the amortized complexity of zero knowledge protocols for multiplicative relations. In Adam Smith, editor, *ICITS 12: 6th International Conference on Information Theoretic Security*, volume 7412 of *Lecture Notes in Computer Science*, pages 62–79, Montreal, QC, Canada, August 15–17, 2012. Springer, Heidelberg, Germany.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Heidelberg, Germany.

[CF85] Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme (extended abstract). In *26th Annual Symposium on Foundations of Computer Science*, pages 372–382, Portland, Oregon, October 21–23, 1985. IEEE Computer Society Press.

[CFG+20a] Alessandro De Carli, Muriel Figueredo Franco, A. Gassmann, Christian Killer, Bruno Rodrigues, Eder J. Scheid, D. Schoenbaechler, and Burkhard Stiller. Wetrace - A privacy-preserving mobile COVID-19 tracing approach and application. *CoRR*, abs/2004.08812, 2020.

[CFG+20b] Justin Chan, Dean P. Foster, Shyam Gollakota, Eric Horvitz, Joseph Jaeger, Sham M. Kakade, Tadayoshi Kohno, John Langford, Jonathan Larson, Sudheesh Singanamalla, Jacob E. Sunshine, and Stefano Tessaro. PACT: privacy sensitive protocols and mechanisms for mobile contact tracing. *CoRR*, abs/2004.03544, 2020.

[CGHZ16] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 998–1021, Hanoi, Vietnam, December 4–8, 2016. Springer, Heidelberg, Germany.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *Advances in Cryptology – EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118, Konstanz, Germany, May 11–15, 1997. Springer, Heidelberg, Germany.

[Cha83] David Chaum. Blind signature system. In David Chaum, editor, *Advances in Cryptology – CRYPTO'83*, page 153, Santa Barbara, CA, USA, 1983. Plenum Press, New York, USA.

[Cha88] David Chaum. Blind signature systems. U.S. Patent #4,759,063, July 1988.

[Cha20] Chaos Computer Club. 10 requirements for the evaluation of "contact tracing" apps. https://www.ccc.de/en/updates/2020/contact-tracing-requirements, 2020. Accessed: 2020-08-23.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[CHRT20] Andrew Clement, Jilian Harkness, George Rain, and Laura Tribe. Snowden surveillance archive. *https://snowdenarchive.cjfe.org/greenstone/cgi-bin/library.cgi*, 2020.

[CKKZ20] Michele Ciampi, Nikos Karayannidis, Aggelos Kiayias, and Dionysis Zindros. Updatable blockchains. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020: 25th European Symposium on Research in Computer Security, Part II*, volume 12309 of *Lecture Notes in Computer Science*, pages 590–609, Guildford, UK, September 14–18, 2020. Springer, Heidelberg, Germany.

[CKL+20] Ran Canetti, Yael Tauman Kalai, Anna Lysyanskaya, Ronald L. Rivest, Adi Shamir, Emily Shen, Ari Trachtenberg, Mayank Varia, and Daniel J. Weitzner. Privacy-preserving automated exposure notification. *IACR Cryptol. ePrint Arch.*, 2020:863, 2020.

[CL99a] Miguel Castro and Barbara Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. Technical Report MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[CL99b] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[CNG18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly blockchain. *CoRR*, abs/1812.11747, 2018.

[Cor] The Corda Platform. https://www.r3.com/corda-platform/.

[Cor20] Corona-Warn Team. Criteria for the evaluation of contact tracing apps. https://github.com/corona-warn-app/cwa-documentation/blob/ec703906c109bd7c3cc84bc361b7e703b20650ea/pruefsteine.md, 2020. Accessed: 2020-08-23.

[CWA+09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[D'A83] Nino D'Angelo. Pronto si tu. https://www.youtube.com/watch?v=8DP3UyDS0Ts, 1983.

[DCCD+19] Claudio Di Ciccio, Alessio Cecconi, Marlon Dumas, Luciano García-Bañuelos, Orlenys López-Pintado, Qinghua Lu, Jan Mendling, Alexander Ponomarev, An Binh Tran, and Ingo Weber. Blockchain support for collaborative business processes. *Informatik Spektrum*, 42:182–190, May 2019.

[DD18] Evan Duffield and Daniel Diaz. Dash: A payments-focused cryptocurrency, 2018. https://github.com/dashpay/dash/wiki/Whitepaper.

[Dec19] Decred. Decred white paper, 2019. https://docs.decred.org/.

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 66–98, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[DMP88] Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge proof systems. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 52–72, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Heidelberg, Germany.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX*, pages 303–320, 2004.

[DMT19] Dominic Deuber, Bernardo Magri, and Sri Aravinda Krishnan Thyagarajan. Redactable blockchain in the permissionless setting. In *2019 IEEE Symposium on Security and Privacy*, pages 124–138, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[DP-20] DP-3T Team. Decentralized privacy-preserving proximity tracing. *https://github.com/DP-3T/documents/blob/master/DP3T%20White%20Paper.pdf*, 2020.

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse Merkle trees. In *NordSec 2016*, volume 10014 of *LNCS*, pages 199–215. Springer, 2016.

[DR20a] Paul-Olivier Dehaye and Joel Reardon. Proximity tracing in an ecosystem of surveillance capitalism. *CoRR*, abs/2009.06077, 2020.

[DR20b] Paul-Olivier Dehaye and Joel Reardon. Swisscovid: a critical analysis of risk assessment by swiss authorities. *CoRR*, abs/2006.10719, 2020.

[DRZ18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2028–2041, 2018.

[DT20a] DP-3T's Team. DESIRE: A Practical Assessment. https://github.com/DP-3T/documents/blob/master/Security%20analysis/DESIRE%20-%20A%20Practical%20Assessment.pdf, 2020. Accessed: 2020-06-01.

[DT20b] DP-3T's Team. Privacy and Security Risk Evaluation of Digital Proximity Tracing Systems. https://github.com/DP-3T/documents/blob/master/Security%20analysis/Privacy%20and%20Security%20Attacks%20on%20Digital%20Proximity%20Tracing%20Systems.pdf, 2020. Accessed: 2020-04-21.

[DT20c] DP-3T's Team. Response to 'Analysis of DP3T: Between Scylla and Charybdis'. https://github.com/DP-3T/documents/blob/master/Security%20analysis/Response%20to%20'Analysis%20of%20DP3T'.pdf, 2020. Accessed: 2020-04-23.

[DT20d] DP-3T's Team. Secure upload authorisation for digital proximity tracing. https://github.com/DP-3T/documents/blob/master/DP3T%20-%20Upload%20Authorisation%20Analysis%20and%20Guidelines.pdf, 2020. Accessed: 2020-05-03.

[EGSR16] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, Santa Clara, CA, 2016.

[EOS19] EOS Canada: What is the role of a block producer? https://www.eoscanada.com/en/what-is-the-role-of-a-block-producer, 2019.

[Eth] Ethereum. http://ethereum.org.

[Eur20] European Commission. Guidance on apps supporting the fight against COVID 19 pandemic in relation to data protection. *Official Journal of the European Union*, 2020.

[FHBS19] Martin Florian, Sebastian Henningsen, Sophie Beaucamp, and Björn Scheuermann. Erasing data from blockchain nodes. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 367–376. IEEE, 2019.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 152–168, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Fra20] Fraunhofer AISEC. Pandemic contact tracing apps: Dp-3t, pepp-pt ntk, and robert from a privacy perspective. Cryptology ePrint Archive, Report 2020/489, 2020. https://eprint.iacr.org/2020/489.

[GAG+19] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 568–580, 2019.

[Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture*

Notes in Computer Science, pages 626–645, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[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*, pages 51–68. ACM, 2017.

[GK20] Juan A. Garay and Aggelos Kiayias. SoK: A consensus taxonomy in the blockchain era. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 284–318, San Francisco, CA, USA, February 24–28, 2020. Springer, Heidelberg, Germany.

[GKL15a] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[GKL15b] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[GKLP18] Juan A. Garay, Aggelos Kiayias, Nikos Leonardos, and Giorgos Panagiotakos. Bootstrapping the blockchain, with applications to consensus and fast PKI setup. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 465–495, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 698–728, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[GKR18] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. Cryptology ePrint Archive, Report 2018/248, 2018. https://eprint.iacr.org/2018/248.

[GKW+16] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 3–16, 2016.

[GKZ19] Peter Gazi, Aggelos Kiayias, and Dionysis Zindros. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy*, pages 139–156, San Francisco, CA, USA, May 19–23, 2019. IEEE Computer Society Press.

[GL20] Mattew Green and Yehuda Lindell. Privacy & tracking to mitigate pandemics: politics and technological solutions. *https://www.brighttalk.com/webcast/17700/392003/privacy-tracking-to-mitigate-pandemics-politics-and-technological-solutions*, 2020.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 581–612, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[Goo14] L.M Goodman. Tezos —a self-amending crypto-ledger white paper, 2014. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.

[Goo20a] Google. Exposure Key export file format and verification. https://developers.google.com/android/exposure-notifications/exposure-key-file-format, 2020. Accessed: 2020-08-23.

[Goo20b] Google. Exposure Notification Cryptography Specification. https://blog.google/documents/69/Exposure_Notification_-_Cryptography_Specification_v1.2.1.pdf, 2020. Accessed: 2020-08-23.

[Gre19] Andy Greenberg. The clever cryptography behind apple's 'find my' feature. *https://www.wired.com/story/apple-find-my-cryptography-bluetooth/*, 2019.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326, Vienna, Austria, May 8–12, 2016. Springer, Heidelberg, Germany.

[gRP] gRPC. http://grpc.io.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[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.

[Imm20] Immuni Team. Immuni's high-level description. https://github.com/immuni-app/immuni-documentation, 2020. Accessed: 2020-08-23.

[IPT20a] Inria PRIVATICS Team. DESIRE: A Third Way for a European Exposure Notification System. https://github.com/3rd-ways-for-EU-exposure-notification/project-DESIRE/blob/master/DESIRE-specification-EN-v1_0.pdf, 2020. Accessed: 2020-06-03.

[IPT20b] Inria PRIVATICS Team. ROBERT: ROBust and privacy-presERving proximity Tracing. https://github.com/ROBERT-proximity-tracing/documents/blob/master/ROBERT-specification-EN-v1_0.pdf, 2020. Accessed: 2020-05-02.

[IVV21] Vincenzo Iovino, Serge Vaudenay, and Martin Vuagnoux. On the effectiveness of time travel to inject covid-19 alerts. In *Proc. of CT-RSA*, volume To appear, 2021.

[JKS16] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of Gyges: Investigating the future of criminal smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 283–295, Vienna, Austria, October 24–28, 2016. ACM Press.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

[Ker20] Thomas Kerber. Implementations to accompany "mining for privacy". GitHub, 2020. https://github.com/tkerber/pistis-impl.

[Kil95] Joe Kilian. Improved efficient arguments (preliminary version). In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 311–324, Santa Barbara, CA, USA, August 27–31, 1995. Springer, Heidelberg, Germany.

[KJG+16] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 279–296. USENIX Association, 2016.

[KJG+18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598, 2018.

[KKK19] M. Kim, Y. Kwon, and Y. Kim. Is stellar as secure as you think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 377–385, 2019.

[KKK21] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Mining for privacy: How to bootstrap a snarky blockchain. *Financial Cryptography and Data Security 2021*, 2021.

[KKL+18] Aggelos Kiayias, Annabell Kuldmaa, Helger Lipmaa, Janno Siim, and Thomas Zacharias. On the security properties of e-voting bulletin boards. In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 505–523, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, 1997.

[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 *2016 IEEE Symposium on Security and Privacy*, pages 839–858, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498, Tokyo, Japan, March 3–6, 2013. Springer, Heidelberg, Germany.

[KRDO17a] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[KRDO17b] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[KZM+15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. https://eprint.iacr.org/2015/1093.

[L. 16] L. Baird. The Swirlds Hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf, 2016.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[Lau83] Mariano Laurenti. La Discoteca. https://www.youtube.com/watch?v=t9kwU27FG7U, 1983.

[LF20a] Dough Leith and Stephen Farrell. Testing apps for COVID-19 tracing (TACT). https://down.dsg.cs.tcd.ie/tact/, 2020. Accessed: 2020-08-23.

[LF20b] Douglas J. Leith and Stephen Farrell. Coronavirus contact tracing: evaluating the potential of using bluetooth received signal strength for proximity detection. *Comput. Commun. Rev.*, 50(4):66–74, 2020.

[LHML20] Franck Legendre, Mathias Humbert, Alain Mermoud, and Vincent Lenders. Contact tracing: An overview of technologies and cyber risks. *CoRR*, abs/2007.02806, 2020.

[LK17] Kevin Liao and Jonathan Katz. Incentivizing blockchain forks via whale transactions. In *Financial Cryptography*, pages 264–279, 2017.

[LLM+19] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 80–96, New York, NY, USA, 2019. Association for Computing Machinery.

[LNZ+16] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 17–30, 2016.

[LPGBD+19] Orlenys López-Pintado, Luciano García-Bañuelos, Marlon Dumas, Ingo Weber, and Alexander Ponomarev. Caterpillar: A business process execution engine on the Ethereum blockchain. *Software: Practice and Experience*, 47(7):1162–1193, Jul 2019.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[MAK13] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 2111–2128. ACM Press, November 11–15, 2019.

[MBS13] Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in byzantine-tolerant state machine replication. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*, pages 61–70, 2013.

[MHH+18] Roman Matzutt, Jens Hiller, Martin Henze, Jan Henrik Ziegeldorf, Dirk Müllmann, Oliver Hohlfeld, and Klaus Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. In Sarah Meiklejohn and Kazue Sako, editors, *FC 2018: 22nd International Conference on Financial Cryptography and Data Security*, volume 10957 of *Lecture Notes in Computer Science*, pages 420–438, Nieuwpoort, Curaçao, February 26 – March 2, 2018. Springer, Heidelberg, Germany.

[MHM18] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. Smart contracts for bribing miners. In *Financial Cryptography*, pages 3–18, 2018.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[MMZ+20] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. CanDID: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. Cryptology ePrint Archive, Report 2020/934, 2020. https://eprint.iacr.org/2020/934.

[MWA+18] Jan Mendling, Ingo Weber, Wil Aalst, Jan vom Brocke, Cristina Cabanillas, Florian Daniel, Søren Debois, Claudio Di Ciccio, Marlon Dumas, Schahram Dustdar, Avigdor Gal, Luciano García-Bañuelos, Guido Governatori, Richard Hull, Marcello La Rosa, Henrik Leopold, Frank Leymann, Jan Recker, Manfred Reichert, and Liming Zhu. Blockchains for business process management—challenges and opportunities. *ACM Transactions on Management Information Systems*, 9(1), Feb 2018.

[MXC+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 31–42, 2016.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[NIS02] NIST. Secure hash standard. https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2withchangenotice.pdf, 2002. Accessed: 2020-08-13.

[NKW21] Tejaswi Nadahalli, Majid Khabbazian, and Roger Wattenhofer. Timelocked bribing. In *Financial Cryptography*, volume To appear, 2021.

[PAC20] PACT's Team. Decentralized privacy-preserving proximity tracing. *https://pact.mit.edu/wp-content/uploads/2020/04/The-PACT-protocol-specification-ver-0.1.pdf*, 2020.

[PDC17] Ivan Puddu, Alexandra Dmitrienko, and Srdjan Capkun. μchain: How to forget without hard forks. Cryptology ePrint Archive, Report 2017/106, 2017. https://eprint.iacr.org/2017/106.

[Ped91] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In Donald W. Davies, editor, *Advances in Cryptology – EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526, Brighton, UK, April 8–11, 1991. Springer, Heidelberg, Germany.

[PEP20] PEPP-T's Team. Pan-european privacy-preserving proximity tracing. *https://www.pepp-pt.org/*, 2020.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[Pie20a] Krzysztof Pietrzak. Delayed authentication: Preventing replay and relay attacks in private contact tracing. *IACR Cryptology ePrint Archive*, 2020:418, 2020.

[Pie20b] Krzysztof Pietrzak. Delayed authentication: Preventing replay and relay attacks in private contact tracing. In Karthikeyan Bhargavan, Elisabeth Oswald, and Manoj Prabhakaran, editors, *Progress in Cryptology - INDOCRYPT 2020: 21st International Conference in Cryptology in India*, volume 12578 of *Lecture Notes in Computer Science*, pages 3–15, Bangalore, India, December 13–16, 2020. Springer, Heidelberg, Germany.

[PL20] Protocol Labs. Ipfs. https://ipfs.io/, 2020. Accessed: 2020-05-05.

[PR20] Benny Pinkas and Eyal Ronen. Hashomer - a proposal for a privacy-preserving bluetooth based contact tracing scheme for hamagen. https://github.com/eyalr0/HashomerCryptoRef/blob/master/documents/hashomer.pdf, 2020. Accessed: 2020-04-27.

[pri20] Report on privacy-enhancing cryptographic protocols for ledgers. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, 2020. http://priviledge-project.eu/publications/deliverables.

[pri21] Revision of privacy-enhancing cryptographic primitives for ledgers. PRIViLEDGE project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020, 2021. http://priviledge-project.eu/publications/deliverables.

[PSHW20] Christoph Prybila, Stefan Schulte, Christoph Hochreiner, and Ingo Weber. Runtime verification for business processes utilizing the Bitcoin blockchain. *Future Generation Computer Systems*, 107:816–831, Jun 2020.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[RGK20] Adam Krellenstein Rosario Gennaro and James Krellenstein. Exposure notification system may allow for large-scale voter suppression. https://static1.squarespace.com/static/5e937afbfd7a75746167b39c/t/5f47a87e58d3de0db3da91b2/1598531714869/Exposure_Notification.pdf, 2020. Accessed: 2020-08-23.

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1759–1776. ACM Press, November 11–15, 2019.

[Sch90a] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch90b] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany.

[Sch18] Berry Schoenmakers. MPyC secure multiparty computation in Python. GitHub https://github.com/lschoe/mpyc, 2018.

[Sei20] Otto Seiskari. Contact Tracing BLE sniffer PoC. https://github.com/oseiskar/corona-sniffer, 2020. Accessed: 2020-06-02.

[Sem20] Semaphore Team. Semaphore. https://semaphore.appliedzkp.org/, 2020. Accessed: 2020-09-15.

[SSL20] TU DARMSTADT SYSTEM SECURITY LAB, CYSEC. TraceCORONA: Anonymous distributed contact tracing for pandemic response. https://tracecorona.net/, 2020. Accessed: 2020-09-05.

[SST20] Joosep Simm, Jamie Steiner, and Ahto Truu. Verifiable multi-party business process automation. In *BPM 2020: Business Process Management Workshops, Proceedings*, volume 397 of *LNBIP*, pages 30–41. Springer, 2020.

[STV+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *IEEE Symposium on Security and Privacy,*

SP 2016, San Jose, CA, USA, May 22-26, 2016, pages 526–545. IEEE Computer Society, 2016.

[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, *ACNS 16: 14th International Conference on Applied Cryptography and Network Security*, volume 9696 of *Lecture Notes in Computer Science*, pages 346–366, Guildford, UK, June 19–22, 2016. Springer, Heidelberg, Germany.

[SWB19] Josh Swihart, Benjamin Winston, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. ECC Blog, February 2019. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/.

[Swi20a] Swiss Federal Office of Public Health. New coronavirus: Swisscovid app and contact tracing. https://www.bag.admin.ch/bag/en/home/krankheiten/ausbrueche-epidemien-pandemien/aktuelle-ausbrueche-epidemien/novel-cov/swisscovid-app-und-contact-tracing/datenschutzerklaerung-nutzungsbedingungen.html#-11360452, 2020. Accessed: 2020-08-23.

[Swi20b] Swiss National Cyber Security Center. Security issue submission [inr-4434]. detailed analysis. https://www.melani.admin.ch/dam/melani/de/dokumente/2020/INR-4434_NCSC_Risk_assessment.pdf.download.pdf/INR-4434_NCSC_Risk_assessment.pdf, 2020. Accessed: 2020-08-23.

[Swi20c] Swiss National Cyber Security Center. Swisscovid proximity tracing system - public security test. https://www.melani.admin.ch/dam/melani/de/dokumente/2020/SwissCovid_Public_Security_Test_Current_Findings.pdf.download.pdf/SwissCovid_Public_Security_Test_Current_Findings.pdf, 2020. Accessed: 2020-08-23.

[Tan20a] Qiang Tang. Privacy-preserving contact tracing: current solutions and open questions. *CoRR*, abs/2004.06818, 2020.

[Tan20b] Qiang Tang. Privacy-preserving contact tracing: current solutions and open questions. Cryptology ePrint Archive, Report 2020/426, 2020. https://eprint.iacr.org/2020/426.

[TBM+20] Sri Aravinda Krishnan Thyagarajan, Adithya Bhat, Bernardo Magri, Daniel Tschudi, and Aniket Kate. Reparo: Publicly verifiable layer to repair blockchains. *arXiv preprint arXiv:2001.00486*, 2020.

[TCN20] TCNCoalition. TCN Protocol. https://github.com/TCNCoalition/TCN#the-tcn-protocol, 2020. Accessed: 2020-05-03.

[Ten] Tendermint. http://tendermint.com.

[TJS16] Jason Teutsch, Sanjay Jain, and Prateek Saxena. When cryptocurrencies mine their own business. In *Financial Cryptography*, pages 499–514, 2016.

[TLW18] An Binh Tran, Qinghua Lu, and Ingo Weber. Lorikeet: A model-driven engineering tool for blockchain-based business process execution and asset management. In *BPM 2018*, volume 2196 of *CEUR Workshop Proceedings*, pages 56–60. CEUR-WS.org, 2018.

[Tra] TraceTogether - behind the scenes look at its development process. https://www.tech.gov.sg/media/technews/

tracetogether-behind-the-scenes-look-at-its-development-process. Accessed: 2020-05-02.

[Vau20a] Serge Vaudenay. Analysis of DP3T. *IACR Cryptol. ePrint Arch.*, 2020:399, 2020.

[Vau20b] Serge Vaudenay. Centralized or decentralized? the contact tracing dilemma. *IACR Cryptol. ePrint Arch.*, 2020:531, 2020.

[VCBL09] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine Fault Tolerance with a spinning primary. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, 2009.

[vRS04] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[VTL17] Yaron Velner, Jason Teutsch, and Loi Luu. Smart contracts make bitcoin mining pools vulnerable. In *Financial Cryptography*, pages 298–316, 2017.

[Vuk15] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security (iNetSec)*, pages 112–125, 2015.

[VV20] Serge Vaudenay and Martin Vuagnoux. Analysis of swisscovid. https://lasec.epfl.ch/people/vaudenay/swisscovid/swisscovid-ana.pdf, 2020. Accessed: 2020-08-23.

[Wik] Wikipedia. Bullrun (decryption program). *https://en.wikipedia.org/wiki/Bullrun_(decryption_program)*.

[Wil18] Zachary J. Williamson. Aztec. https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf, 2018. Accessed: 2020-09-15.

[WLT+19] I. Weber, Q. Lu, A. B. Tran, A. Deshmukh, M. Gorski, and M. Strazds. A platform architecture for multi-tenant blockchain-based systems. In *ICSA 2019*, pages 101–110. IEEE, 2019.

[Woo16] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2016.

[WW19] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.

[Yan19] H. Yang. *EC Cryptography Tutorials - Herong's Tutorial Examples*. Herong's Tutorial Examples. Herong Yang, 2019.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 347–356, 2019.

[zca] Zcash. https://z.cash/.

[Zca18] Zcash. Parameter generation. https://z.cash/technology/paramgen/, 2018.

[Zca19] Zcash. Address and value pools in Zcash. https://zcash.readthedocs.io/en/latest/rtd_pages/addresses.html#turnstiles, 2019.

[ZCC+16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 270–282, Vienna, Austria, October 24–28, 2016. ACM Press.

[ZkD20] ZkDAI Team. Zkdai. https://github.com/atvanguard/ethsingapore-zk-dai, 2020. Accessed: 2020-09-15.

[ZMM+20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. DECO: Liberating web data using decentralized oracles for TLS. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20: 27th Conference on Computer and Communications Security*, pages 1919–1938, Virtual Event, USA, November 9–13, 2020. ACM Press.

[ZOB19] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. A treasury system for cryptocurrencies: Enabling better collaborative intelligence. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24-27, 2019. The Internet Society.

[ZoK20] ZoKrates Team. Zokrates. https://zokrates.github.io/, 2020. Accessed: 2020-09-15.