# Scaling Distributed Ledgers and Privacy-Preserving Applications

Alberto Sonnino

A dissertation submitted in partial fulfillment

of the requirements for the degree of

**Doctor of Philosophy**

of

**University College London**.

Department of Computer Science

University College London

January 25, 2021

I, Alberto Sonnino, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

# Abstract

This thesis proposes techniques aiming to make blockchain technologies and smart contract platforms practical by improving their scalability, latency, and privacy. This thesis starts by presenting the design and implementation of Chainspace, a distributed ledger that supports user defined smart contracts and execute user-supplied transactions on their objects. The correct execution of smart contract transactions is publicly verifiable. Chainspace is scalable by sharding state; it is secure against subsets of nodes trying to compromise its integrity or availability properties through Byzantine Fault Tolerance (BFT). This thesis also introduces a family of replay attacks against sharded distributed ledgers targeting cross-shard consensus protocols; they allow an attacker, with network access only, to double-spend resources with minimal efforts. We then build Byzcuit, a new cross-shard consensus protocol that is immune to those attacks and that is taillored to run at the heart of Chainspace. Next, we propose FastPay, a high-integrity settlement system for pre-funded payments that can be used as a financial side-infrastructure for Chainspace to support low-lantency retail payments. This settlement system is based on Byzantine Consistent Broadcast as its core primitive, foregoing the expenses of full atomic commit channels (consensus). The resulting system has extremely low-latency for both confirmation and payment finality. Finally, this thesis proposes Coconut, a selective disclosure credential scheme supporting distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. It ensures authenticity and availability even when a subset of credential issuing authorities are malicious or offline, and natively integrates with Chainspace to enable a number of scalable privacy-preserving applications.

# Impact Statement

The work in this thesis can inform the design of new and existing projects that implement distributed ledgers, smart contract platforms or applications, in order to increase their scalability, security, and privacy. Chainspace (Chapter 3) and Coconut (Chapter 6) are used as part of DECODE (DEcentralised Citizen-owned Data Ecosystems) [1], a European project with a digital democracy pilot in Barcelona implementing a decentralised petitions platform. Chainspace was also commercialised in a company (`chainspace.io`) co-founded by the author of this thesis, and the team was acquired by Facebook. Coconut was also integrated into Cosmos SDK [2] and commercialised in a company (Nym [3]) aiming to provide an open-ended anonymous overlay network that disguise patterns in internet traffic; it uses the scheme described in Chapter 6 as anonymous authentication credentials to enable privacy-enhanced data transfer and decentralized identity. Byzcuit and the work done on replay attacks against sharded distributed ledgers (Chapter 4) have profound impact on the security of recently proposed systems such as Omniledger [4] and RapidChain [5]; these systems were presented at top security conferences and form the basis of numerous start-ups and open-source projects such as Harmony [6]. Finally, the content of this thesis has been presented on multiple occasions, both at academic venues and industry conferences; some of its chapters have been published at top-tier security conferences. It produced multiple tools and open-source software, and is freely available online.

# Acknowledgements

This work would not have been possible without my primary supervisor George Danezis, who helped me throughout the past few years of research, and to whom I wish to express my profound gratitude. Special thanks to my secondary supervisors Jens Groth and Ioannis Psaras for their unwaivering support and generous encouragement, and to my close collaborators, Mustafa Al-Bassam and Shehar Bano, who have been the source of many fruitful discussions. I have been privileged to have had the opportunity to work with many brilliant and helpful people around the world. Specifically, I thank all my co-authors (in alphabetic order) Christos Andrikos, Sarah Azouvi, Lejla Batina, Mathieu Baudet, Vitalik Buterin, Avery Ching, Lukasz Chmielewski, Andrey Chursin, Dave Hrycyszyn, Ismail Khoffi, Michał Król, Liran Lerman, Zekun Li, Dahlia Malkhi, Vasilis Mavroudis, Sarah Meiklejohn, Patrick McCorry, Kostas Papagiannopoulos, Dmitri Perelman, Guilherme Perin, Giorgos Rassias, Etienne Rivière, Argyrios Tasiopoulos, Lixia Zhang, and Zhiyi Zhang. I would also like to thank Ramsey Khoury, my other co-founder at `chainspace.io` not mentioned above, and all my other former colleagues at `chainspace.io`: Penny Andrews, Andy Bennett, Stuart Chinery, Jérémy Letang, and Lola Oyelayo-Pearson. I also thank everyone at Facebook Novi, specifically David Marcus, James Everingham, Christian Catalini, Kevin Weil, Ben Maurer, Morgan Beller, and Evan Cheng for providing a supportive working environment. Finally, I would like to thank the European Commission for funding my research with a PhD scholarship, and all my friends and family for their continuous support and encouragement.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

> The root problem with conventional currency is all the trust that's required to make it work. The central bank must be trusted not to debase the currency, but the history of fiat currencies is full of breaches of that trust. Banks must be trusted to hold our money and transfer it electronically, but they lend it out in waves of credit bubbles with barely a fraction in reserve. We have to trust them with our privacy, trust them not to let identity thieves drain our accounts.
>
> ---
>
> *Satoshi Nakamoto*

## 1.1 Problem Statement

Blockchains lie at the foundation of Bitcoin and other cryptocurrencies, which have a total global market capital of over $450B as of November 2020 [7]. The blockchain is a decentralized, replicated, immutable and tamper-evident log: data on the blockchain cannot be deleted, and anyone can read data from the blockchain and verify its correctness. The blockchain is maintained by a set of authorities (called *nodes*) that form a distributed network. An important implication of this architecture is *disintermediation*: multiple untrusted or semi-trusted parties can *directly* and *transparently* interact with each other without the presence of a trusted intermediary. This makes blockchains immediately relevant to banks and financial institutions which incur huge middleman costs in settlements and other back office operations. A number of big players are actively exploring the feasibility of blockchains, including the Bank of England [8], the Bank of America [9] and the IMF [10]. In addition to

the financial industry, blockchains have been employed in a diverse array of use cases, ranging from voting [11], through data storage [12], to the sharing economy [13, 14]. Despite their useful properties and applications, adoption of blockchains is nowhere near as ubiquitous as their traditional counterparts due to their performance limitations. These properties are deeply related to the *consensus* protocol—the core component of the blockchain.

Consensus protocols are defined by two key properties. The first is related to performance, and requires that requests from correct clients are eventually processed (*liveness*). The second property is related to security, and states that if an honest node accepts (or rejects) a value then all other honest nodes make the same decision (*safety/consistency*). A plethora of consensus protocols exist that offer different trade-offs between the two key properties of consensus liveness and consistency. The distributed systems community has extensively studied consensus for over two decades, and developed robust and practical protocols that can tolerate faulty and malicious nodes [15, 16]. However, these protocols were designed for closed groups, and cannot be readily adapted to blockchains.

Bitcoin's fundamental innovation was to enable consensus among nodes forming a peer-to-peer network [17]. This was achieved via a leader election based on proof-of-work (PoW): all nodes attempt to find the solution to a hash puzzle and the node that wins adds the next block to the blockchain. Due to its probabilistic leader election process combined with performance fluctuations in decentralized networks, Bitcoin offers only weak consistency: different nodes might end up having different views of the blockchain leading to *forks*. Additionally, Bitcoin suffers from poor performance and its PoW consumes a huge amount of energy [18]. Bitcoin's underlying blockchain technology suffers from scalability issues: with a current block size of 1MB and 10 minute inter-block interval, throughput is capped at about 7 transactions per second, and a client that creates a transaction has to wait for about 10 minutes to confirm. In contrast, mainstream payment processing companies like Visa confirm transactions within a few seconds, and have high throughput of over 24,000 transactions per second [19]. Reparametrization of Bitcoin—such as

Bitcoin-NG [20]—can improve this to a limited extent up to 27 transactions per second and 12 second latency, respectively [21]. More significant improvement requires a fundamental redesign of the blockchain paradigm. This has led to an array of proposals for new systems and new consensus protocols [22].

Unlike application specific blockchain technologies, such as Bitcoin for a currency, or certificate transparency [23] for certificate verification, smart contract platforms like Ethereum [24] introduce a design that offers extensibility allowing nodes to execute user-defined programs on transactions. Traditionally, users submit transactions to the blockchain, which are sequenced through consensus, executed by the nodes, and permanently stored on the blockchain; all transactions and smart contracts are public to allow anyone to verify the correctness of the blockchain. This paradigm makes it hard to build privacy-preserving applications on top of blockchains as neither the smart contracts logic or the transactions can contain secret values. The restriction of not being able to include secret values inside transactions and smart contracts is extremely limiting. Even a simple smart contract as 'digitally sign a document' is problematic since the signing key must be kept secret, and therefore cannot be part of the transaction or smart contract.

## 1.2 Overview

This thesis proposes technologies to make blockchains practical. It specifically aims to overcome the following limitations: (i) poor scalability, (ii) high latency, and (iii) difficulty to operate on secret values (privacy). Figure 1.1 presents a high level overview of the system presented in this thesis, that is built from the following main ingredients: Chainspace, Byzcuit, FastPay, and Coconut.

**Chainspace, a scalable backbone with integrated privacy support**. The main component of Figure 1.1 is Chainspace that constitutes the backbone of the system and allows it to scale to accommodate high throughput. Chainspace is a sharded distributed ledger; sharding is one of the main approaches to address blockchain scalability issues. The key idea is to create groups (called *shards*) of nodes that handle only a subset of all transactions and system state, relying on classical Byzantine Fault

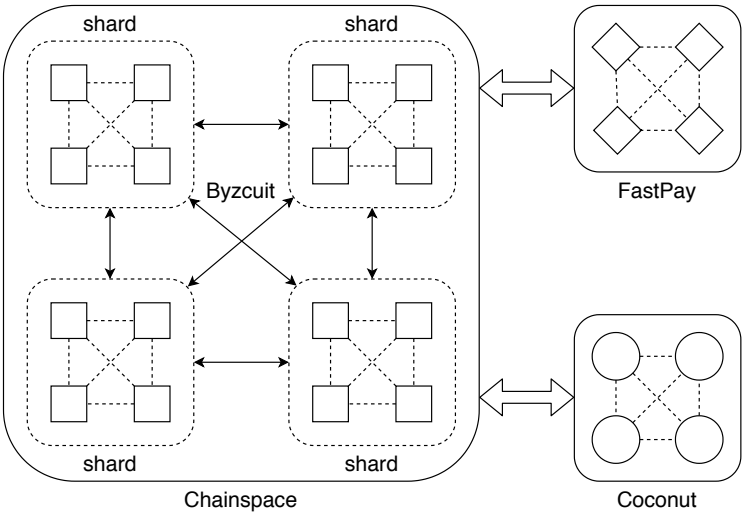


**Figure 1.1:** Global overview.

Tolerance (BFT) protocols for reaching consensus amongst each shard. Sharded systems achieve great throughput and linear scalability because: (i) non-conflicting transactions can be processed in parallel by multiple shards; and (ii) the system can scale up by adding new shards. That is, the throughput of the system is theoretically unbounded as it can be arbitrarily increased by adding new shards. However, this separation of transaction handling across shards is not perfectly 'clean'—a transaction might rely on data managed by multiple shards, requiring an additional step of *cross-shard consensus* across the concerned shards. To that purpose Chainspace runs Byzcuit at its core, a new cross-shard consensus protocol.

Orthogonally to its sharded design, Chainspace is a smart contract platform supporting privacy-preserving applications by design; it revisits the execution of smart contracts on blockchains, and proposes a system where the transaction is executed by the client, and the smart contract platform only verify the correctness of the execution.

**FastPay, a low-latency payment system**. Sharded blockchains can be a solid backbone for financial systems, but their latency bottleneck remains their underlying BFT consensus protocol which makes them unpractical for retail payment at physical points of sale. To overcome this issue, we present a side-infrastructure called FastPay. FastPay is a distributed settlement system for pre-funded payments that can

be used as a financial side-infrastructure to support low-latency retail payments of a primary system such as Chainspace. This side-infrastructure complements scalable (high-throughput) blockchains by achieving extremely low latency by foregoing the expenses of consensus, making the system applicable to point of sale payments.

**Coconut, privacy-preserving credentials for smart contract applications**. Coconut is a novel selective disclosure credential scheme that natively integrates with blockchains by distributing the issuance phase across a set of authorities. Coconut allows to issue privacy-preserving digital identities to users without relying on a trusted third party. We then leverage those credentials along with the new privacy-preserving execution model of Chainspace to build a number of decentralised and scalable privacy-persevering applications as Chainspace smart contracts.

## 1.3 Dissertation Organisation and Contributions

This dissertation is organized as follows. Chapter 2 lays the foundations on which the core chapters rely. Chapter 3 presents Chainspace, a novel sharded smart contract platform that supports user defined smart contract, and that scales by sharding state and the execution of transactions through a new cross-shard atomic commit protocol. Chapter 4 presents the first replay attacks on cross-shard consensus in sharded blockchains, describes the issues that lead to these vulnerabilities, and presents Byzcuit, a novel cross-shard consensus protocol that is immune to those attacks. Chapter 6 presents Coconut, a credential system allowing permissioned and semi-permissioned blockchains to issue credentials through smart contracts.

This dissertation has resulted in the following publications (in chronological order, relevant chapter indicated in bold):

> Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. In *Network and Distributed System Security Symposium*, 2018

**(Chapter 3)** Chainspace is a decentralized infrastructure, known as a distributed ledger, that supports user defined smart contracts and executes user-supplied transactions on their objects. The correct execution of smart contract transactions is

verifiable by all. The system is scalable, by sharding state and the execution of transactions. Chainspace is secure against subsets of nodes trying to compromise its integrity or availability properties through Byzantine Fault Tolerance (BFT), and extremely high-auditability, non-repudiation and 'blockchain' techniques. Even when BFT fails, auditing mechanisms are in place to trace malicious participants. We present the design, rationale, and details of Chainspace; we argue about its scaling and other features; we illustrate a number of privacy-friendly smart contracts for smart metering, polling and banking and measure their performance.

Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. In *Network and Distributed Systems Security Symposium*, 2019

**(Chapter 6)** Coconut is a novel selective disclosure credential scheme supporting distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. Coconut integrates with blockchains to ensure confidentiality, authenticity and availability even when a subset of credential issuing authorities are malicious or offline. We implement and evaluate a generic Coconut smart contract library for Chainspace and Ethereum; and present three applications related to anonymous payments, electronic petitions, and distribution of proxies for censorship resistance. Coconut uses short and computationally efficient credentials, and our evaluation shows that all Coconut cryptographic primitives can be executed in less than 10 milliseconds on a commodity laptop.

Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the Age of Blockchains. In *ACM Conference on Advances in Financial Technologies*, 2019

**(Chapter 2)** The core technical component of blockchains is *consensus*: how to reach agreement among a distributed network of nodes. A plethora of blockchain

consensus protocols have been proposed—ranging from new designs, to novel modifications and extensions of consensus protocols from the classical distributed systems literature. The inherent complexity of consensus protocols and their rapid and dramatic evolution makes it hard to contextualize the design landscape. We address this challenge by conducting a systematization of knowledge of blockchain consensus protocols. After first discussing key themes in classical consensus protocols, we describe: (i) protocols based on proof-of-work; (ii) proof-of-X protocols that replace proof-of-work with more energy-efficient alternatives; and (iii) hybrid protocols that are compositions or variations of classical consensus protocols. This survey is guided by a systematization framework we develop, to highlight the various building blocks of blockchain consensus design, along with a discussion on their security and performance properties. We identify research gaps and insights for the community to consider in future research endeavours.

Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay Attacks and Defenses Against Cross-shard Consensus in Sharded Distributed Ledgers. In *IEEE European Symposium on Security and Privacy*, 2020

**(Chapter 4)** We present a family of replay attacks against various sharded distributed ledgers targeting cross-shard consensus protocols. They allow an attacker, with network access only, to double-spend or lock resources with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks can also exhibit themselves as faults under periods of asynchrony. These attacks are effective against both shard-led and client-led cross-shard consensus approaches. We present Byzcuit—a new cross-shard consensus protocol that is immune to those attacks. We implement a prototype of Byzcuit and evaluate it on a real cloud-based testbed, showing that our defenses impact performance minimally, and overall performance surpasses previous works.

Mathieu Baudet, George Danezis, and Alberto Sonnino. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *ACM Conference on Advances in Financial Technologies*, 2020

**(Chapter 5)** FastPay allows a set of distributed authorities, some of which are Byzantine, to maintain a high-integrity and availability settlement system for pre-funded payments. It can be used to settle payments in a native unit of value (crypto-currency), or as a financial side-infrastructure to support retail payments in fiat currencies. FastPay is based on Byzantine Consistent Broadcast as its core primitive, foregoing the expenses of full atomic commit channels (consensus). The resulting system has low-latency for both confirmation and payment finality. Remarkably, each authority can be sharded across many machines to allow unbounded horizontal scalability. Our experiments demonstrate intra-continental confirmation latency of less than 100ms, making FastPay applicable to point of sale payments. In laboratory environments, we achieve a throughput of over 80,000 transactions per second with 20 authorities—surpassing the requirements of current retail card payment networks, while significantly increasing their robustness.

## 1.4 Additional Work

Below are other publications outside scope did while doing a PhD, listed for inclusion in chronological order:

1. Mustafa Al-Bassam, Alberto Sonnino, Michał Król, and Ioannis Psaras. Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations. arXiv preprint arXiv:1805.06411, 2018

2. Michał Król, Alberto Sonnino, Mustafa Al-Bassam, Argyrios Tasiopoulos, and Ioannis Psaras. Proof-of-Prestige: A Useful Work Reward System for Unverifiable Tasks. In *IEEE International Conference on Blockchain and Cryptocurrency*, 2019

3. George Danezis and Alberto Sonnino. SybilQuorum: Open Distributed Ledgers Through Trust Networks. arXiv preprint arXiv:1906.12237, 2019

4. Alberto Sonnino, Michał Król, Argyrios G Tasiopoulos, and Ioannis Psaras. AStERISK: Auction-based Shared Economy ResolutIon System for blocKchain. In *Workshop on Decentralized IoT Systems and Security*, 2019

5. Alberto Sonnino. FMPC: Secure Multiparty Computation from Fourier Series and Parseval's Identity. arXiv preprint arXiv:1912.02583, 2019

6. Christos Andrikos, Lejla Batina, Lukasz Chmielewski, Liran Lerman, Vasilios Mavroudis, Kostas Papagiannopoulos, Guilherme Perin, Giorgos Rassias, and Alberto Sonnino. Location, location, location: Revisiting modeling and exploitation for location-based side channel leakages. In *Asiacrypt*, 2019

7. Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Privacy-preserving, Asynchronous Single Sign-On. International Symposium on Privacy Enhancing Technologies, 2021

8. Michał Król, Alberto Sonnino, Argyrios G. Tasiopoulos, Ioannis Psaras, and Etienne Rivière. PASTRAMI: Privacy-preserving, Auditable, Scalable & Trustworthy Auctions for Multiple Items. In *ACM/IFIP Middleware*, 2020

9. Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: White-Glove Approach for BFT Testing. arXiv preprint arXiv:2004.10617, 2020

10. Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities. In *Financial Cryptography and Data Security*, 2021

## 1.5 Work Done in Collaboration

A large part of this work has been conducted in collaboration with other researchers. All the coauthors contributed to the development of the work listed in Section 1.3.

The data model (Section 3.1.1) and application interface (Section 3.2) of Chainspace is a joint work with all the co-authors of the paper; and I led the design

(Section 3.5), implementation and evaluation (Section 3.6) of the Chainspace's system and application smart contracts. I led the design of the replay attacks against cross-shard consensus protocols presented in Sections 4.2 and 4.3, including the techniques to elicit messages to replay (Section 4.4); I also led the design of Byzcuit (Section 4.6), and the security proofs showing that Byzcuit is immune to those attacks (Section 4.6.2). The design of FastPay (Section 5.3) is a joint work with all the co-authors of the paper, and I performed its evaluation (Section 5.6). I led the cryptographic construction of Coconut (Section 6.2), as well as its implementation (Section 6.4) and benchmark (Section 6.6); I also led the design and implementation of its smart contract library (Section 6.4.1) and privacy-preserving smart contract applications (Section 6.5) on Chainspace.

This work would not have been possible without my co-authors. Specifically, Mustafa Al-Bassam led the implementation of the Chainspace smart contract framework (Section 3.6) and the Coconut smart contract library on Ethereum (Section 6.5.1). Mustafa Al-Bassam and Shehar Bano led the implementation and evaluation of Byzcuit (Section 4.7). Mathieu Baudet led the security proofs (Section 5.4) and the implementation (Section 5.5) of FastPay; and Sarah Meiklejohn wrote the security proofs of the Coconut threshold credentials scheme (Section 6.3).

# Chapter 2

# Background and Related Works

This chapter introduces the terminology and assumptions used throughout this thesis, and provides context to explain the reasons that led to the design choices of Chainspace (Chapter 3) and Byzcuit (Chapter 4). It also provides background on the algorithm at the core of FastPay (Chapter 5), and on the cryptogarphic building blocks used by Coconut (Chapter 6). More niche related works are presented in the appropriate chapters.

## 2.1 Terminology and Assumptions

This section presents basic concepts, terminology and assumptions related to consensus and blockchains. We refer readers interested in detailed, formal consensus definitions to the work by Garay and Kiayias [37].

**Consensus**. The consensus protocol enables a distributed network of nodes to agree on the total order of some input values [38]. In the blockchain context, consensus helps reach agreement on whether *transactions* should be accepted or rejected, and in which order. A transaction specifies some transformation on the blockchain state. If a transaction passes validity and verification checks (*transaction validation*), it is included in a candidate *block* (a set of transactions) to be added to the blockchain.

**Permissioned vs. permissionless blockchains**. In *permissioned blockchains* [39, 40], identities of all the nodes that run consensus are known (trusted or semi-trusted), and their admission is controlled by a single entity or federation. In *permissionless blockchains* [17], anyone can run a node and join the network. Permissioned

blockchains sometimes imply limited write access; in this work, we only refer to its meaning within the context of consensus, as defined earlier.

**Consistency**. The fact that a network of $n$ nodes reaches consensus on a proposed value; it can be either strong [15, 25, 4, 41] or weak [17, 24, 42, 43]. In strong consistency, the shared state across honest nodes does not diverge. In weak consistency, the shared state across nodes might diverge temporarily leading to *forks*, and additional mechanisms are needed for reconciling forks. This is related to *eventual consistency—i.e.* the blockchain becomes consistent eventually. *Finality* refers to the guarantee that a block will be permanently added to the blockchain.

**Properties**. We consider *liveness* and *safety* as enumerated by Cachin *et al.* [44]. For liveness, *validity* ensures that if a node broadcasts a message, eventually this message will be ordered within the consensus, and *agreement* ensures that if a message is delivered to one honest node, it will eventually be delivered to all honest nodes. For safety, *integrity* guarantees that only broadcast messages are delivered, and they are delivered only once, and *total order* ensures that all honest nodes extract the same order for all delivered messages.

**Synchrony assumptions**. Networks may be *synchronous* or *asynchronous*, or offer *eventual synchrony* [45]. In a *synchronous* network the delays messages may suffer can be bound by some time $\Delta$. On the other hand, in *asynchronous* networks messages may be delayed arbitrarily, and there exists no reliable bound $\Delta$ for their delay. Networks with partial synchrony (or eventual synchrony, or semi-synchronous networks) assume that the network will become and remain synchronous at Global Stabilization Time (GST) despite potentially a long period of asynchrony.

**Network propagation**. Consensus protocols make certain assumptions about how messages propagate across nodes within the network. In *point-to-point channels* [38], there is a pairwise connection between all nodes which is both reliable and authenticated. In the peer-to-peer (p2p) messaging model, a node 'diffuses' a message into the network, which is expected to eventually reach all honest nodes with some probability [36]. Every node knows a set of other nodes (*peers*)—when a message is

received, nodes diffuse it by passing it on to their peers. A node may not be aware of the identities or number of other nodes in the network. *Gossip-based protocols* [46] rely on this assumption by considering that each node has a point-to-point connection with at least a subset of the network; the size of that subset is a security parameter.

**Communication complexity**. The communication complexity of a consensus protocol refers to the maximum number of messages exchanged between the nodes in a single run of the consensus protocol. Note that a single run might involve multiple rounds of message exchanges before it completes (*i.e.* consensus is reached).

**Performance**. The performance of consensus protocols is usually defined in terms of *throughput* (*i.e.* the maximum rate at which values can be agreed upon by the consensus protocol), *scalability* (*i.e.* the system's ability to achieve greater throughput when consensus involves a larger number of nodes) and *latency* (*i.e.* the time it takes from when a value is proposed, until when it is totally ordered).

**Adversary model for consensus**. The adversary model is the fraction of malicious or faulty nodes that the consensus protocol can tolerate (*i.e.* it will operate correctly despite the presence of such nodes). This is usually referred to as the *failure model* in the distributed systems literature. In the *crash failure* model, nodes may fail at any time—but they fail by stopping to process, emit or receive messages. Usually failed nodes remain silent forever, although a number of distributed protocols consider recovery. On the other hand, in the *byzantine failures* model, failed nodes may take arbitrary actions—including sending and receiving sequences of messages that are specially crafted to defeat properties of the consensus protocol. Another failure model in the context of consensus protocols relates to *network partition*: when network devices fail (or are attacked) such that the network splits into two or more relatively independent subnets.

**Adversary model for blockchain consensus**. Blockchain consensus has extended the adversarial model to include several new threats. In consensus protocols with weak consistency guarantees, nodes might end up having different views of the blockchain (*forks*) because of latency in propagation of transactions, and faulty or

malicious nodes. A related concept is that of *double-spending* where a transaction consumes an asset which has already been consumed by a previous transaction. *DoS resistance* defines resilience of the node(s) involved in consensus to denial-of-service (DoS) attacks. In the context of permissionless blockchains, *Sybil attacks* refer to an attacker's ability to create fake identities or subvert existing nodes, and take over majority of the network [47]. Sybil attacks assume two types of adversary models: static and adaptive. A *static adversary* has corrupted a fixed number of nodes in advance—it cannot corrupt new nodes or create new identities over time. An *adaptive adversary* has flexibility in the nodes it can corrupt and the new identities it can create over time, to improve its probability of controlling majority of the network.

**Decentralization**. This is a key property of the blockchain that enables a number of other properties such as censorship resistance, attack resistance and fault-tolerance. Decentralization has no formal definition, but generally [48] refers to a system that: (i) is run by multiple machines and has no architectural choke point (*architectural decentralization*); (ii) is run by multiple independent individuals or organisations (*political decentralization*); and (iii) comprises multiple interfaces and data structures that can fully operate independently, instead of acting as a single whole (*logical decentralization*). Section 2.2 discusses the impact of different consensus design choices on decentralization; however, a detailed discussion is beyond the scope of this work and we refer interested readers to the work by Troncoso *et al.* [49].

## 2.2 Consensus in the Age of Blockchains

This section summarises the evolution of blockchain consensus starting from Bitcoin [17] up to state-of-the-art sharded systems. This provides context and explains the reasons that led to the design choices of Chainspace (Chapter 3) and Byzcuit (Chapter 4), and provides background on the core of FastPay (Chapter 5).

Systems like Bitcoin [17, 24, 43] probabilistically elect a single node which can extend the blockchain; they assume synchrony for safety, have probabilistic finality (*i.e.*, forks can exist and be eventually accepted) and low performance (*i.e.*, high latency and low throughput). As explained in Chapter 1, Bitcoin suffers from

scalability issues: with a current block size of 1MB and 10 minute inter-block interval, throughput is capped at about 7 transactions per second, and a client that creates a transaction has to wait for about 10 minutes for confirmation. For those reasons, the community shifted to committee-based designs [22] where a group of nodes collectively extends the blockchain typically *via* classical Byzantine fault tolerance (BFT) consensus protocols such as PBFT [50]. While these systems offer better performance, single-committee consensus is not scalable—as every node handles every transaction, adding more nodes to the committee decreases throughput due to the increased communication overhead. This motivated the design of *sharded* systems, where multiple committees handle a subset of all the transactions—allowing parallel execution of transactions.

### 2.2.1 Classical Consensus

Consensus protocols have been studied in the distributed systems community since the 1970s [51]. These protocols were intended for closed, small groups of nodes. We provide an overview of key themes in classical consensus literature, with the goal to contextualize the rest of the section. We will revisit some of these concepts when discussing committee-based consensus (Sections 2.2.3 and 2.2.4).

**Two-Phase commit**. Jim Gray, in 1978, proposed the two-phase commit protocol [52], allowing a *transaction manager* to atomically commit a transaction, depending on different resources held by a distributed set of servers called *resource managers*. Transaction commit protocols enable distributed processing, and thus scalability—but do not provide resilience against faulty resource managers, or more generally nodes. In fact, two-phase commit suffers a deadlock in case a resource manager fails to complete the protocol, requiring the introduction of more complex three-round protocols allowing recovery [53]—*i.e.* the distributed resource managers being able to release the locks held on resources. Since potentially a crucial resource may only be available on a single resource manager, any failures inhibit progress towards accepting dependent transactions. Chapter 4 combines this primitive with Byzantine agreement in a novel way to design a scalable consensus protocol.

**Consensus, atomic broadcast and state machine replication**. The need for consensus, or atomic broadcast protocols, in distributed systems originates from the need to provide resilience against failures across multiple nodes holding *replicas* of a database. *Atomic broadcast* [54] allows a set of servers to agree on a value associated with an instance of the protocol; and *consensus protocols* extend this to agreeing on a sequence of values. This primitive is closely associated with the state machine replication paradigm [55] for building reliable distributed computations: any computation is expressed as a state machine, accepting messages to mutate its state. Given that a set of replicas start at the same initial state, and can agree on a common sequence of messages, then they may all locally evolve the state of the computation and correctly maintain consistency across the replicated databases they hold, despite failures or network variations. The underlying consensus protocols are characterized by the communication model, as well as the failure model, assumed (Section 2.1). Fischer *et al.* [56] show that deterministic protocols for consensus are impossible in the fully asynchronous case, and have known solutions in the synchronous case (also known as the "Byzantines General's Problem" [57]).

**Key protocols**. In the network security literature Byzantine nodes would be considered malicious or collectively controlled by an adversary. Thus the Byzantine setting is of relevance to security-critical settings, and traditional consensus protocols tolerating only crash failures such as Paxos [16], viewstamped replication [58], Raft [59], or Zab [60] cannot be used, unmodified, in adversarial settings. Practical Byzantine Fault Tolerance (PBFT) by Castro and Liskov [15] is the canonical protocol implementing consensus in the the Byzantine and eventually synchronous setting.

**Byzantine consistent broadcast**. Byzantine consistent broadcast is a protocol allowing a sender to broadcast a message to nodes while guaranteeing the following properties in the presence of Byzantine faults: *validity*, *no duplication*, *integrity*, and *consistency* [38]. Validity means that if the sender is honest, all honest nodes eventually deliver the message; no duplication ensures that every honest node delivers at most one message; integrity means that nodes deliver a message only if it originates from the sender (*i.e.* the sender is not impersonated); and consistency

ensures that if two honest nodes deliver a message, it is the same message (no honest nodes deliver different messages). The properties of the protocol are guaranteed if $n = 3f + 1$, where $n$ is the total number of nodes and $f$ is the number of Byzantine nodes. Byzantine consistent broadcast can be implemented in a number of ways, the most notable is called *Signed Echo Broadcast* which works in three steps [38]. In the first step, the sender disseminates a digitally signed message to all nodes (best effort broadcast), then all honest nodes witness the message by replying with a signed acknoledgement. Finally, the sender collects these signed acknowledgements and relays them in a third communication step to all nodes. Byzantine consistent broadcast is the primitive at the core of FastPay (Chapter 5).

**Practical Byzantine Fault Tolerance (PBFT)**. PBFT operates in a sequence of views, each coordinated by a leader—a pattern also used in Paxos [16]. Within each view the leader orders messages, and propagates them through a three-step reliable broadcast to the replicas. Replicas monitor the leader for safety, as well as for liveness, and can propose a *view change* in case the leader is unavailable or malicious. Safety is guaranteed within the asynchronous network setting; liveness on the other hand is only guaranteed within a partially synchronous setting, since replicas rely on time-outs to detect a faulty leader. The key complexity of PBFT lies in the view-change sub-protocol, that needs to ensure agreement on the new leader and view, as well as guarantee safety of messages agreed in previous views. The basic protocol requires $\mathcal{O}(n^2)$ messages for $n$ replicas to achieve consensus, where $n$ is the number of nodes. The properties of the protocol are guaranteed if $n = 3f + 1$, where $f$ is the number of Byzantine nodes. Hotstuff [61] is one of the latest successors of PBFT. It builds on a body of works [62, 63] to achieve a number of improvements over PBFT; notably, it operates with a communication complexity of $\mathcal{O}(n)$ messages and has a simpler view change protocol.

**Limitations of classical consensus**. PBFT and other consensus protocols employ replication to achieve resilience against failures, not scalability. In fact the traditional literature on Byzantine consensus does not discuss distribution of resources, in the context of a distributed or sharded database, with the exception of a less known joint

work by Gray and Lamport on combining atomic broadcast with atomic commit [64]. As a result, one expects systems employing Byzantine consensus to see this protocol become a bottleneck, since its trivial application would require all transactions to be sequenced by the quorum of $n$ nodes—using protocols that are slower than asking a single processor to sequence them. Chapter 4 discusses sharded consensus, where multiple quorums only handle a subset of the transactions.

### 2.2.2 Elected Leader Consensus

The need to achieve consensus in open, decentralized networks motivated the design of protocols based on *elected leaders* that write to the blockchain. This may involve a combination of steps, usually applied sequentially: (i) *selection resource* refers to selecting a set of nodes based on some resource they own, for example *via* mining power in proof-of-work (*e.g.* Bitcoin [17]), stakes in proof-of-stake (*e.g.* Cardano [65, 66, 67]), trusted hardware etc.; and (ii) *selection mechanism* refers to a technique that is used to non-deterministically elect the leader. This typically takes the form of a cryptographic *lottery—e.g.* a random beacon, a periodically generated pseudo-random number, which allows the nodes to determine if they have been elected as the leader. We briefly describe PoW and proof-of-Stake consensus, and refer to Bano *et al.* [22] for further details.

**Proof-of-Work consensus**. Proof-of-Work consensus protocols rely on a computational puzzle to elect a leader that writes to the blockchain. As finding a solution to the puzzle requires a significant amount of computational work, so a valid solution is considered to be a proof-of-work (PoW). PoW was first presented by Dwork and Naor in 1993 [68] as a technique for combatting spam mail, by requiring the email sender to compute the solution to a mathematical puzzle to prove that some computational work was performed [69]. PoW was independently proposed in 1997 for Hashcash by Back, another system for fighting spam [70]. In 2008, Bitcoin [17] was published by a pseudonymous author Satoshi Nakamoto. Their key innovation is the use of PoW as a sybil-resistance mechanism, combined with a rule to choose between different versions of the blockchain (fork-choice rule), to achieve consensus—originator—in an open, permissionless network. It was not until 2015—

7 years after Bitcoin was first released—that it was formally proved that Bitcoin PoW is a consensus protocol [71]. While the technical components of Bitcoin originate in previous literature [72], their composition in Bitcoin to achieve consensus is novel.

Nakamoto consensus probabilistically elects a single node which can extend the blockchain: it is based on a PoW puzzle derived from Hashcash [70], which requires finding a hash of a block that is less than a target integer value $t$. As the hashing algorithm is pre-image resistant, the puzzle can be solved only by including random nonces in the block until the resulting hash is valid (*i.e.* less than $t$). The difficulty of the puzzle is therefore adjustable: decreasing $t$ increases the number of guesses (and thus work) required to generate a valid hash. The nodes that generate hashes are called *miners* and the process is referred to as *mining*. Miners calculate hashes of candidate blocks of transactions to be added to the blockchain. Nakamoto consensus relies on the cryptographic paradigm of provers and verifiers. Miners take on the role of provers who mint blocks, and every other node is a verifier who validates (and potentially rejects) blocks according to a list of globally agreed consensus rules. This is the 'trust, but verify' paradigm.

Nakamoto consensus is a fork-tolerant protocol as all nodes reach eventual consistency about the blockchain's content, whereas classical consensus focuses on fork-avoidance protocols as nodes must have a consistent view after every epoch. A fork occurs if two miners find two different blocks that build on the same previous block. An attacker must have sufficient computing power to be able to create a fork of the blockchain that has more accumulated work than the chain that is to be overridden. Thus the threat model assumes an adversary that has the majority of the computing power on the network (referred to as a *51% attack*). The *security threshold* of the network is the percentage of computing power required to conduct a 51% attack. Nakamoto consensus resolves forks by accepting the 'longest chain, which has the greatest PoW effort invested in it' as the correct one. However, such systems assume synchrony, have probabilistic finality (*i.e.*, forks can exist and be eventually accepted) and low performance (*i.e.*, high latency and low throughput).

**Proof-of-Stake consensus**. One of the biggest criticisms of Bitcoin is that it is based

on power-intensive PoW that has no external utility. In proof-of-stake, participants vote on new blocks weighted by their in-band investment such as the amount of currency held in the blockchain (*stake*). A number of systems have provably secure proof-of-stake protocols [67, 66, 73]. A common theme in these systems is to randomly elect a leader from among the stakeholders (participants) *via* lottery, which then appends a block to the blockchain. Leader election may be public, that is the outcome is visible to all the participants [66, 73]. Alternatively, in a private election the participants use private information to check if they have been selected as the leader, which can be verified by all other participants using public information [67]. Leader election based on private lottery is resilient to DoS attacks because participants privately check if they are elected before revealing it publicly in their blocks, at which point it is too late to attack them.

The nature of the lottery varies across different systems, but broadly it is either collaborative (*i.e.* requires coordination between the participants) or independent. In Ouroboros [66], the participants (a random subset of all stakeholders) run a multiparty coin-tossing protocol to agree on a random seed. The participants then feed this seed to a pseudo-random function defined by the protocol, that elects the leader from among the participants in proportion to their stake. The same random seed is used to elect the next set of participants for the next epoch. In Ouroboros Praos [67] and Snow-White [73] participants independently determine if they have been elected. Snow-White selects participants for each epoch based on the previous state of the blockchain, who independently check if they have been elected as the leader. Snow-White uses similar criteria for leader election as Bitcoin, that is finding a pre-image that produces a hash below some target. However, participants are limited to compute only one hash per time step (assuming access to a weakly synchronized clock) and the target takes into account each participant's amount of stake. In Ouroboros Praos, participants generate a random number using a verifiable random function (VRF). If the random number is below a threshold, it indicates that the participant has been elected as the leader, who then broadcasts the block along with the associated proof generated by the VRF to the network. Ouroboros

and Ouroboros Praos distribute rewards among all the participants regardless of whether or not they win the election. PoW's leader election eligibility is out-of-band, and all nodes verify the leader election's result only so far as to find the longest and heaviest chain. Whereas in proof-of-stake the entire leader-election protocol transcript is recorded in-band which increases the nodes' storage, bandwidth and validation overhead for every block.

A challenge for proof-of-stake systems is to keep track of the changing stakes of the stakeholders. Ouroboros requires that shift in stakes is bounded, meaning the statistical distance is limited over a certain number of epochs. Additionally, Snow-White looks at stakes sufficiently far back in time to ensure that everyone has agreed on the stake distribution. Outside academia, some deployed cryptocurrencies incorporate proof-of-stake (*e.g.* Peercoin), but their designs have not been rigorously studied. Ethereum Foundation has been considering using proof-of-stake for some time [74], and some systems like EOS [75] use delegated proof-of-stake, where participants elect delegates of their choice for mining. proof-of-stake systems are subject to a number of attacks. In long-range attacks [76], old nodes' signature keys are compromised and used to re-write the blockchain history; in stake bleeding attacks [77], an adversary accumulates the rewards associated with creation of new blocks in a forking chain in order to inflate its stake and eventually accumulate enough to confirm an inconsistent fork; in nothing-at-stake attacks [63] nodes mine on every fork of the chain to benefit from the rewards of whichever fork wins.

### 2.2.3 Hybrid Consensus: Single Committee

The elected leader approach suffers from poor performance as well as safety limitations such as weak consistency and low fault-tolerance. This has resulted in a shift towards consensus protocols where a *committee*—rather than a single node—collectively drives the consensus.

**Intra-Committee consensus protocol**. The intra-committee consensus protocol ensures that the committee members reach agreement on state of the blockchain. A number of committee-based blockchains [4, 25, 78] use PBFT. The messaging complexity of PBFT's MAC-authenticated all-to-all communication is $O(n^2)$. This

is problematic for permissionless blockchains where a committee can potentially have thousands of nodes. Another approach to reach intra-committee consensus is based on gossip protocols. These protocols are suited to permissionless blockchains as point-to-point connections between the $n$ nodes of the committee are no longer needed. Upon reception of a new transaction, nodes query a subset of $k$ randomly selected others nodes; each of those nodes replies with its view of the state of the system, and initiates a similar query. The requesting node weights the replies and potentially updates its own view of the state; this process is repeated until global consensus is reached (with high probability). Avalanche [46] proposes a gossip-based family of BFT protocols that have a communication complexity of $O(k \times n)$, where $k << n$ is a security parameter. These protocols are leaderless and claim strong Denial of Service (DoS) and censorship resistance.

**Committee leadership**. Traditional BFT protocols proceed in rounds, where consensus in each round is led by a committee leader. The concept of a committee leader is not compatible with permissionless blockchains that aspire to achieve the design goal of decentralization. An adversary can concentrate its DoS attack on committee leaders which are easy to discover by joining the committee. As the leader is responsible for proposing transactions, a malicious leader can prioritize transactions from which it can benefit. While committee members can potentially detect a malicious leader and trigger leader re-election (*i.e.* view change), this severely degrades performance [79]. In Solidus, the leader is external to the committee and can propose transactions and PoW to nominate itself as a committee member only once to the committee. If the committee agrees, they approve the proposed transactions and allow the miner to join the committee in the next round. The proposal, that has now become a decision, also serves as the next puzzle and is propagated to all miners. Solidus highlights a safety problem in PBFT's 'stable' leader which can potentially manipulate reconfiguration by waiting for a malicious miner to solve the puzzle, and later nominating it on to the committee—allowing the committee to gradually become dominated by corrupt members. The concept of a leader in committee-based blockchains introduces a number of challenges with respect to

scalability, and security (DoS attack, transaction censorship, and centralization). This has motivated the design of leaderless consensus protocols [80, 81].

**Optimizations**. A number of optimizations have been proposed to improve the performance of BFT consensus protocols. *Scheduling optimizations* involve techniques to identify and execute non-conflicting transactions in parallel (and thus achieve high throughput) by leveraging application-specific information [82]. *Execution optimizations* reduce latency by allowing clients [83] or replicas [84] to speculatively execute transactions based on predicted results—if a fault is detected (*i.e.* the speculation turns out to be incorrect), the client/replica rolls back its state to the last checkpoint and re-executes the transactions based on the correct results. *Protocol optimizations* refer to the committee's ability to switch between suitable BFT protocols according to varying network conditions and performance requirements [85]. Hyperledger uses *pluggable and modular* consensus in which the consensus protocol can be specified by the smart contract policy. *Cryptographic optimizations* leverage advances in cryptography to optimize the communication complexity of BFT. ByzCoin organizes the consensus committee into a communication tree that uses a primitive called scalable collective signing [86] which reduces PBFT's messaging complexity to $O(n)$. The XFT [87] protocol, on the other hand, improves the efficiency of consensus by relaxing the threat model. It considers that Byzantine nodes may act arbitrarily, however links between honest nodes are reliable and eventually synchronous. This leads to a simplification of the view change and steady state BFT protocol. *Hardware optimizations* enable consensus protocols to achieve high performance by exploiting advances in hardware. The Intel Sawtooth lake system uses the Intel SGX and related trusted execution environments to perform the duties related to ordering transactions, while ensuring safety and liveness [88]. Finally, *architectural optimizations* improve performance by distributing different consensus duties across independent subsets of replicas. A useful paradigm (employed by Hyperledger) is to separate ordering from execution [89], which allows for a modular design where transaction validation is performed by the fully trusted nodes (or endorsers) while the semi-trusted nodes (ordering nodes) order the transactions and

add these to the blockchain. In Hyperledger, clients first submit their transactions to the endorsers who execute the smart contract. A transaction is only submitted to a subset of endorsers according to the policy of the respective smart contract. As different smart contracts can designate different endorsers, execution can take place in parallel. Clients collect matching signed results and smart contract state updates from sufficient number of endorsers, and submit these to the ordering nodes which append it to the blockchain using a consensus protocol. Others [90] argue that distributed ledgers can decouple the ordering—performed in public on cryptographic commitments of transactions—from the validation containing private information, that is only checked by a trusted cabal. Separating ordering from execution [89] allows committee-based blockchains to scale at the same rate as the core ordering protocol, but providing universal end-to-end verifiability and decentralization in this setting remains an open challenge.

### 2.2.4 Hybrid Consensus: Multiple Committees

Single-committee consensus is not scalable and adding more nodes to the committee decreases throughput—leading to the design of consensus based on multiple committees. Transactions are split among multiple committees (called *shards*) which then process these transactions in parallel. Every committee has its own blockchain and set of objects (or unspent transaction outputs, UTXO) that they manage. Committees run an *intra-shard consensus protocol* (*e.g.*, PBFT) within themselves, and extend their blockchain in parallel. This incurs additional coordination between the committees *via inter-committee consensus* to reach agreement on a value among nodes across multiple committees. The inter-committee consensus protocol may be run entirely by the committees (*non-mediated*), or may be mediated by an external party (*mediated*). *Inter-committee configuration* defines how nodes are assigned to the committees in a multiple committees setting; it can be static or dynamic. When multiple committees are involved in consensus, an important consideration is how they will be organized in terms of *topology*.

**Committee topology**. Chainspace (presented in Chapter 3) and Omniledger [4] have flat topologies, that is all committees are at the same level. Elastico [78]

has a hierarchical topology in which a number of 'normal' committees validate transactions, and a leader committee orders these transactions and extends the blockchain. In RSCoin [91], a permissioned blockchain, the central bank controls all monetary supply while committees (called mintettes) authorized by the bank validate a subset (shard) of transactions. The transactions that pass validation are submitted to the central bank which adds them to the blockchain. Hierarchical topology facilitates configuration and management of committees in multi-committee blockchains, but undermines decentralization.

**Inter-committee consensus**. In a multi-committee system, some transactions might manipulate state that is handled by different committees. The inter-committee consensus ensures that such transactions are processed consistently and atomically across all the concerned committees. One approach is to mediate the inter-committee consensus protocol *via* the client. Omniledger uses an atomic commit protocol to process transactions across committees (see Section 4.3.1). Client-driven inter-committee consensus protocols make the assumption that clients are incentivized to proceed to the unlock phase. Such incentives may exist in a cryptocurrency application where an unresponsive client will lose its own coins if the inputs are permanently locked, but do not hold for a general-purpose platform where transaction inputs may have shared ownership. Client-driven inter-committee consensus protocols are vulnerable to DoS attack if the client stops participating midway, resulting in the transaction inputs being locked forever. Another approach is to run an atomic commit protocol collaboratively between all the concerned committees. This is achieved by making the entire committees act as resource managers for the transactions they manage (see Section 4.2.1). Inter-committee consensus protocols are relatively immature, and their security has not been rigorously evaluated. For example, Chapter 4 shows the susceptibility of these protocols to replay attacks that allow an attacker to double-spend resources with minimal effort, and without colluding with any nodes. The attacker records a target committee's responses to the consensus protocol, and replays them during another instance of the protocol.

## 2.3 Cross-Shard Consensus Protocols

Section 2.2.4 presented a brief overview of inter-committee consensus; this section provides a deeper background on cross-shard consensus protocols as it is a core component of Chainspace (presented in Chapter 3) and the subject of Chapter 4.

Multi-committee systems are often called *sharded systems*, and each committee is called *shard*. In sharded systems, some transactions may operate on objects handled by different shards, effectively requiring the relevant shards to additionally run a *cross-shard consensus protocol* to enable agreement across the shards. Specifically, if any of the shards relevant to the transaction rejects it, all the other shards should likewise reject the transaction to ensure atomicity.

### 2.3.1 Two-Phases Atomic Commit Protocols

A typical choice for implementing cross-shard consensus is the two-phase atomic commit protocol [52] (see Section 2.2.1). This protocol has two phases which are run by a *coordinator*. In the first *voting* phase, the nodes tentatively write changes locally, lock resources, and report their status to the coordinator. If the coordinator does not receive status message from a node (*e.g.*, because the node crashed or the status message was lost), it assumes that the node's local write failed and sends a rollback message to all the nodes to ensure any local changes are reversed, and locks released. If the coordinator receives status messages from all the nodes, it initiates the second *commit* phase and sends a commit message to all the nodes so they can permanently write the changes and unlock resources. In the context of sharded blockchains, the atomic commit protocol operates on shards (which make the local changes associated with the voting phase *via* an intra-shard consensus protocol like PBFT), rather than individual nodes. A further consideration is who assumes the role of the coordinator; *client-led protocols* rely on the client to take the role of coordinator, and *shard-led protocols* rely on entire shard.

There are currently two key approaches to cross-shard atomic commit protocols. The first approach involves client-led protocols (such as Atomix [4] and RSCoin [91]), where the client acts as a coordinator. These protocols assume that clients are incentivized to proceed to the unlock phase. The second approach involves shard-

led protocols (such as S-BAC [25] and Elastico [78]), where shards collectively assume the role of a coordinator. All the concerned shards collaboratively run the protocol between them. This is achieved by making the entire shard act as a 'resource manager' for the transactions it handles. Chapter 4 describes a family of replay attacks in the context of two representative systems: S-BAC [25] as an example of shard-led protocols (Section 4.2); and Atomix as an example of client-led protocols (Section 4.3); these attacks can compromise both system liveness and safety.

### 2.3.2 Mutex-Based Consensus Protocols

Contrarily to S-BAC and Atomix that achieve cross-shard consensus through a two-phase atomic commit protocol, mutex-based schemes for cross-shard transactions, such as RapidChain [5] and Ethereum's cross-shard 'yanking' proposal [92], adopt mutex-based schemes for transactions that involve objects managed by different shards. The key idea is to require all objects that a transaction reads or writes to be in the same shard (*i.e.*, all locks for a transaction are local to the shard). Cross-shard transactions are enabled by transferring the concerned objects between shards, effectively giving shards a lock on those objects. When shard 1 transfers an object to shard 2, shard 1 includes a transfer 'receipt' in its blockchain. A client can then send to shard 2 a Merkle proof of this receipt being included in shard 1's blockchain, which makes the object active in shard 2.

Mutex-based schemes also need to consider replay attacks. Clients can claim the same receipt multiple times, unless shards store information about previously claimed receipts. Naïvely, shards have to store information about all previously claimed receipts permanently. However, two intermediate options with trade-offs have been proposed [92]:

- Shards only store information about receipts for $l$ blocks, and clients can only claim receipts within $l$ blocks; objects are permanently lost if not claimed in time (this introduces a synchrony assumption).

- Shards only store information about receipts for $l$ blocks, and include the root of a Merkle tree of claimed receipts in their blockchain every $l$ blocks. If a

receipt is not claimed within $l$ blocks, the client must provide one Merkle proof every $l$ blocks that have passed to show that the receipt has not been previously claimed, in order to claim it. The longer the receipt was not claimed, the greater the number of proofs that are needed to claim a receipt. These proofs need to be also stored on-chain to allow other nodes to validate them.

Chapter 4 presents a system, called Byzcuit, that forgoes the need for shards to store information about old state (such as inactive objects or old receipts) as shards only need to know the set of active objects they manage, and does not impose a trade-off between the amount of information about old state that needs to be stored and the cost of recovering old state that was held up in an incomplete cross-shard transaction (*i.e.*, an unclaimed receipt).

## 2.4 Sybil Resistance and Committee Management

A limitation of using PoW or PoX for sybil resistance in permissionless committees is that the biggest miners will have a greater likelihood of dominating the committee, though at the cost of significantly more hashing power than required for single-leader PoW systems. Other PoX alternatives, relying on space, memory, or space-time, have been proposed but these suffer from similar issues. Protocols have also been proposed for sybil detection based on the analysis of social networks and trust graphs [93], but those have not been adapted to blockchains, besides the definitional framework for Federated Byzantine Agreement Systems proposed by Stellar [94]. Sybilquorum [31] makes a first attempt to bridge this gap by proposing mechanism to detect sybils through a stake-weighted social networks analysis.

### 2.4.1 Sybil Resistance

Committee formation refers to the criteria used to allow nodes to join a committee. *Permissioned* blockchains like Hyperledger [39] operate in a trusted environment where nodes are granted committee membership based on the organizational policy. In permissionless blockchains, the committee is formed so as to thwart sybil attacks. Nodes are usually allowed to join the committee based on a *selection resource* such as *PoW*. In ByzCoin [95], the consensus committee is dynamically formed by a

window of recent miners. Each miner has voting power proportional to its number of mining blocks in the current window, which is proportional to its hash power. When a miner finds a solution to the puzzle, it becomes a member of the committee and receives a share in the consensus. In addition to PoW, Omniledger [4] also supports *proof-of-stake* to allocate committee membership based on directly invested stake. Some permissionless blockchains employ a further *selection mechanism* such as a *lottery* to form the committee. In Algorand [96], all the nodes that have PoX run a verifiable random function—they are promoted to the committee if the output is below a certain value.

**Coercion resistance**. Another consideration for bootstrapping committees is to achieve coercion resistance, in the form of requiring enormous effort for an adversary to suppress the overall operation of the system. Coercion resistance properties are also key to the success of other decentralized systems, such as BitTorrent [97], that are subject to take-down pressures by publishers. Bitcoin itself was coincidentally proposed in 2008 [98], the same year when E-gold [99] was declared illegal by the US Department of Justice and taken offline—illustrating that value exchange systems, and monetary systems that are transnational and unregulated, will come under fire by national monetary and law enforcement authorities. Systems such as Tor [100] have survived in a highly adversarial environment despite parts of its infrastructure, namely directory authorities, being a closed consensus group. These authorities are distributed geographically, and are under different jurisdictions and managed by different organizations to preclude both collusion and single jurisdiction attacks. Single-committee blockchains may, through careful selection of nodes, achieve coercion resistance [22].

**Multiple committees**. Multi-committee blockchains raise the additional issue of how to map nodes to committees. In permissioned systems, the process of assigning nodes to committees is usually done *statically* according to the policy of the federation. Another approach is to *dynamically* allocate nodes to committees. Permissioned systems like RSCoin can use a trusted source of randomness for committee reconfiguration, but this can be problematic in a permissionless setting which

would require a shared random coin [101, 102]. Generating good randomness in a distributed way is a known hard problem: current best solutions tolerate up to $1/6$ fraction of Byzantine nodes, while incurring a high message complexity [103]. Among the more recent solutions, RandHerd [104] provides a more scalable, secure multi-party computation protocol that offers unbiasable, decentralized randomness while tolerating a third of Byzantine faults. It brings down the communication complexity to $O(c^2\log(n))$, where $c$ is the size the subgroups it uses. In multi-committee blockchains, nodes should be assigned to committees in a non-deterministic way to stop an adversary from concentrating its presence in one committee and exceeding the Byzantine-tolerance threshold.

**Securing committees**. The idea of scaling services built on state machine replication (SMR) by splitting state (or sharding) among multiple committees (also called partitions or shards) has been well-studied in the context of traditional distributed systems [101, 102, 105]. The key challenge in these systems is to ensure linearizability by atomically executing operations that span multiple committees. These systems employ fault-tolerant BFT protocols at their core as the nodes are controlled by a single entity or a group of entities that collectively govern the system. Due to similar governance assumptions, these techniques can be extended to permissioned blockchains. Sharding permissionless blockchains with Byzantine adversaries is challenging and tackled by only a few recent systems [25, 4, 78, 5]. Individual committees can tolerate up to 33% of malicious members, otherwise the malicious committee can compromise all the transactions that touch the bad committee.

**Committee governance**. Randomly mapping nodes to committees improves security, but prohibits finer governance. General-purpose platforms like Chainspace might have different policies within committees; for example some committees can be permissioned while others can be permissionless. In this case it might be useful to enforce node-to-shard mapping *via* smart contracts that allow a node to join a committee trusted by the smart contract provider.

### 2.4.2 Committee Reconfiguration

Intra-committee configuration means how nodes are assigned to the committee. In *static* configuration, nodes are statically assigned to the committee, and are allowed to stay on indefinitely. Static configuration is typically employed in permissioned blockchains like Hyperledger and RSCoin. In *dynamic* configuration, committee members are changed periodically. This model is typically used in permissionless blockchains as this helps thwart sybil attacks. Dynamic committee membership can take three forms. (i) In *rolling (single)* membership, the committee is updated in a sliding window fashion, *i.e.* a new node replaces the oldest committee member periodically. In ByzCoin, when a miner finds a solution to the puzzle, it becomes a member of the committee and receives a share in the current consensus window which moves one step forward (ejecting the oldest miner). (ii) *Rolling (multiple)* committee membership is a similar concept, where multiple committee members are replaced periodically. Omniledger uses cryptographic sortition to select a subset of the committee to be swapped out and replaced with new members. (iii) Some systems replace the *full* committee, *e.g.* Algorand and Snow-White select the committee members for each epoch using randomness generated based on previous blocks.

**Liveness in dynamic committee configuration**. Dynamic committee configuration improves security by raising the bar for sybil attacks, but introduces a new challenge: how is liveness maintained during reconfiguration? One approach is to only do rolling configuration, which has the benefit that the committee is operational during reconfiguration as the operational members can continue to process transactions while a fraction of the committee is being reconfigured and bootstrapped. Omniledger uses cryptographic sortition to select a subset of the committee to be swapped out and replaced with new members. This is done in such a way that the ratio between honest and Byzantine members in a committee is maintained. In Solidus [106], a new miner joining the committee can propose transactions only once. This binds transaction proposals to reconfiguration, so it is no longer possible for an old committee to approve transactions concurrent to a reconfiguration event.

**Reconfiguration of multiple committees systems**. Inter-committee configuration

means whether node assignment to committees in a multi-committee blockchain remains *static* or is periodically changed (*dynamic*). Omniledger periodically reconfigures committees to ensure that a committee is never compromised. This is achieved by a secure shard reconfiguration protocol, based on RandHerd, that committee members run periodically and autonomously. In every epoch, a random subset of members is replaced with new set of members that registered their interest in the previous epoch. The swap operation is done such that liveness is maintained during reconfiguration events because a subset of committee members continues to be operational. Dynamic inter-committee configuration prevents an adversary from subverting existing nodes in a committee and exceeding the Byzantine-tolerance threshold. Chainspace has abstracted details of committee reconfiguration and it is up to policy enforced *via* a smart contract to decide how nodes are allocated to committees. Nodes can be added (and removed) to committees by their members through majority voting.

## 2.5 Selective Disclosure Credentials

This section provides some background on selective disclosure credentials and on some useful cryptographic building blocks. Chapter 6 presents Coconut, a novel selective disclosure credential scheme that is used to build a number of decentralised and scalable privacy-preserving applications on top of Chainspace. Selective disclosure credentials (or anonymous credentials) [107, 108, 109] allow the issuance of credentials to users, and the subsequent unlinkable revelation to a verifier. They allow users to be known in different contexts by different pseudonyms; a user might be known to one service under one pseudonym and to another service under a different pseudonym, and yet be unlinkable accross services. Users can selectively disclose some of the attributes embedded in the credential or specific functions of these attributes. At the high level, anonymous credentials scheme have two main protocols: (i) the *issuing* protocol where the issuing authorities provide the user with a credential embedding a number of attributes, and (ii) the *showing* protocol where the user shows its credential (and potentially some of its attributes) to a verifier.

### 2.5.1 Cryptographic Building Blocks

We describe the security properties of anonymous credentials and provide background on zero-knowledge proofs that are extensively used at the core of any anonymous credential scheme. We then present the cryptographic assumptions on which Coconut as well as many of its predecessors (see Section 2.5.2) rely.

**Security properties**. Anonymous credentials scheme satisfy *unforgeability*, *blindness*, and *unlinkability* (or *zero-knowledge*). Unforgeability means it is unfeasible for an adversarial user to convince an honest verifier that they are in possession of a credential if they are in fact not (*i.e.*, if they have not received a valid credential from the issuing authority). Blindness ensures it is unfeasible for an adversarial authority to learn any information about the user attributes during the credentials issuing protocol, except for what is explicitly revealed by the user. Finally, unlinkability (or Zero-knowledge) ensures it is unfeasible for an adversarial verifier (potentially working with an adversarial authority) to learn anything about the user attributes (except for what is explicitly revealed by the user) during the execution of the showing protocol, or to link the execution of the showing protocol with either another execution of that protocol or with the credentials issuing protocol.

**Zero-knowledge proofs**. Zero-knowledge proofs are protocols allowing a *prover* to convince a *verifier* that it knows a secret value $x$, without revealing any information about that value. The prover can also convince the verifier that they know a secret value $x$ satisfying some statements $\phi$. Anonymous credentials extensively employ zero-knowledge proofs to provide users with certified secret values; users are successively able to prove to third party verifiers that they hold secret values certified by specific credentials issuers, and prove statements about those values without disclosing them. This enables, for instance, the property of *provable personal properties*. A credential issuer may provide a user with a secret value $x = 20$ representing their age; the user can then prove in zero-knowledge to a verifier that a specific credential issuer certified that their age is larger than 18, without revealing their real age $x$.

Coconut uses non-interactive zero-knowledge proofs (NIZK) to assert knowledge and relations over discrete logarithm values. These proofs can be efficiently

implemented without trusted setups using sigma protocols [110], which can be made non-interactive using the Fiat-Shamir heuristic [111] in the random oracle model.

**Cryptographic assumptions**. Coconut requires groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order $p$ with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and satisfying the following properties: (i) *Bilinearity* means that for all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$ and $(a,b) \in \mathbb{F}_p^2$, $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$; (ii) *Non-degeneracy* means that for all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, $e(g_1, g_2) \neq 1$; (iii) *Efficiency* implies the map $e$ is efficiently computable; (iv) furthermore, $\mathbb{G}_1 \neq \mathbb{G}_2$, and there is no efficient homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. The type-3 pairings are efficient [112]. They support the XDH assumption which implies the difficulty of the Computational co-Diffie-Hellman (co-CDH) problem in $\mathbb{G}_1$ and $\mathbb{G}_2$, and the difficulty of the Decisional Diffie-Hellman (DDH) problem in $\mathbb{G}_1$ [113]. Coconut also relies on a cryptographically secure hash function $H^*$, hashing an element $\mathbb{G}_1$ into an other element of $\mathbb{G}_1$, namely $H^* : \mathbb{G}_1 \to \mathbb{G}_1$. We implement this function by serializing the $(x,y)$ coordinates of the input point and applying a full-domain hash function to hash this string into an element of $\mathbb{G}_1$ (as Boneh *et al.* [113]).

### 2.5.2 The Predecessors of Coconut

This section describes the body of works on top of which Coconut is built, starting with some relevant signature schemes (which provide unforgeability but neither blindness nor unlinkability, see Section 2.5.1).

**Short and aggregable signatures**. The Waters signature scheme [114] provides the bone structure of our primitive, and introduces a clever solution to aggregate multiple attributes into short signatures. However, the original Waters signatures do not allow blind issuance or unlinkability, and are not aggregable as they have not been built for use in a multi-authority setting. Lu *et al.* scheme, commonly known as LOSSW signature scheme [115], is also based on Waters scheme and comes with the improvement of being sequentially aggregable. In a sequential aggregate signature scheme, the aggregate signature is built in turns by each signing authority; this requires the authorities to communicate with each other resulting in increased latency and cost. The BGLS signature [116] scheme is built upon BLS signatures

and is remarkable because of its short signature size—signatures are composed of only one group element. The BGLS scheme has a number of desirable properties as it is aggregable without needing coordination between the signing authorities, and can be extended to work in a threshold setting [117]. Moreover, Boneh *et al.* show how to build verifiably encrypted signatures [116] which is close to our requirements, but not suitable for anonymous credentials.

**Anonymous credentials**. CL Signatures [107, 118] and Idemix [119] are amongst the most well-known building blocks that inspired applications going from direct anonymous attestations [120, 121] to electronic cash [122]. They provide blind issuance and unlikability through randomization; but come with significant computational overhead and credentials are not short as their size grows linearly with the number of signed attributes, and are not aggregable. U-Prove [123] and Anonymous Credentials Light (ACL) [124] are computationally efficient credentials that can be used once unlinkably; therefore the size of the credentials is linear in the number of unlinkable uses. Pointcheval and Sanders [109] present a construction which is the missing piece of the BGLS signature scheme; it achieves blindness by allowing signatures on committed values and unlinkability through signature randomization. However, it only supports sequential aggregation and does not provide threshold aggregation. For anonymous credentials in a setting where the signing authorities are also verifiers (i.e., without public verifiability), Chase *et al.* [108] develop an efficient protocol. Its 'GGM' variant has a similar structure to Coconut, but forgoes the pairing operation by using message authentication codes (MACs). None of the above schemes support threshold issuance.

While the scheme of Garman *et al.* [125] does not specifically focus on threshold issuance of credentials or on general purpose credentials, it provides the ability to issue credentials without central issuers supporting private attributes, blind issuance, and unlinkable multi-show selective disclosure. To obtain a credential, users build a vector commitment to their secret key and a set of attributes; and append it to a ledger along with a pseudonym built from the same secret key, and a zk-proof asserting the correctness of the vector commitment and of the pseudonym. To show a credential

under a different pseudonym, users scan the ledger for all credentials and build a RSA accumulator; they provide a zk-proof that they know a credential embedded in the accumulator. Similarly to Zerocoin [42], showing credentials requires an expensive double discrete-logarithm proof (about 50KB [125]); and the security of the credentials scheme relies on the security of the ledger. Coconut addresses the two open questions left as future work by Garman *et al.* [125]; (i) the security of Coconut credentials do not depend on the security of a transaction ledger as they are general purpose credentials, and (ii) Coconut enjoys short and efficient proofs as it builds from blind signatures and does not require cryptographic accumulators.

**Short and threshold issuance anonymous credentials**. Coconut (Chapter 6) extends these previous works by presenting a short, aggregable, and randomizable credential scheme; allowing threshold and blind issuance, and a multi-authority anonymous credentials scheme. Coconut primitives do not require sequential aggregation, meaning the aggregate operation does not have to be performed by each signer in turn. Any independent party can aggregate any threshold number of partial signatures into a single aggregate credential, and verify its validity.

# Chapter 3

# Chainspace: A Sharded Smart Contracts Platform

This thesis introduces technologies to make blockchains practical, specifically aiming to overcome the following limitations: (i) poor scalability, (ii) high latency, and (iii) difficulty to operate on secret values (privacy). This chapter starts tackling blockchain scalability and privacy by presenting Chainspace, a scalable backbone system with integrated privacy support. Chainspace is the main component of Figure 1.1 and is highlighted in Figure 3.1. Chainspace is a sharded distributed ledger; sharding is one of the main approaches to address blockchain scalability issues. The key idea is to create groups (called *shards*) of nodes that handle only a subset of all transactions and system state, relying on classical Byzantine Fault Tolerance (BFT) protocols for reaching consensus amongst each shard. Figure 3.1 illustrates an instantiation of Chainspace with four shards containing four nodes each. Nodes are represented by black squares and the dashed lines connecting them to each other indicate they run a BFT protocol. The solid arrows connecting each shards are a *cross-shard consensus protocol* allowing to coordinate shards operations; this protocol is discussed in Section 3.3.3 and is the subject of Chapter 4.

Unlike application specific blockchains, such as Bitcoin [17] for a currency, Google Spanner [126] for a relational database, or certificate transparency [23] for certificate verification, Chainspace offers extensibility though smart contracts, like Ethereum [24]. However, users expose to Chainspace enough information

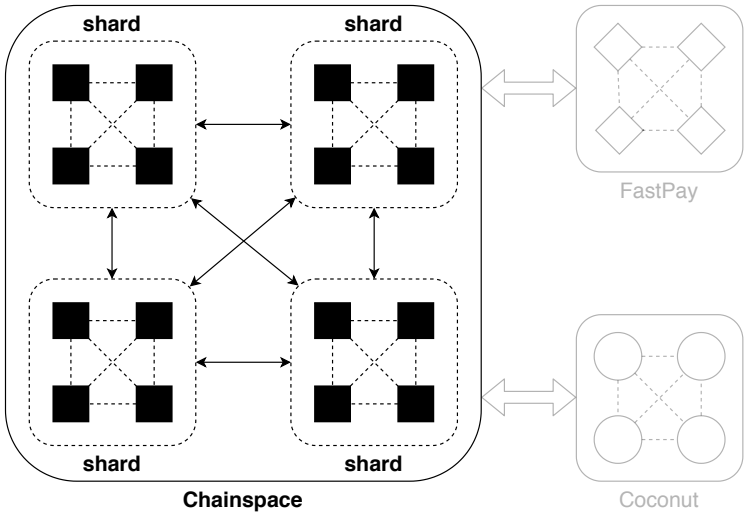


**Figure 3.1:** Global overview: Chainspace.

about contracts and transaction semantics, to provide higher scalability through sharding across infrastructure nodes. Furthermore, our platform is agnostic as to the smart contract language, or identity infrastructure, and supports privacy features through modern zero-knowledge techniques [127, 128]. Chainspace supports privacy-preserving applications by design; it revisits the execution of smart contracts on blockchains, and proposes a system where the transaction is executed by the client, and the smart contract platform only verify the correctness of the execution.

Unlike other scalable but 'permissioned' smart contract platforms, such as Hyperledger Fabric [39] or BigchainDB [129], Chainspace aims to be an 'open' system: it allows anyone to author a smart contract, anyone to provide infrastructure on which smart contract code and state runs, and any user to access calls to smart contracts. Further, it provides ecosystem features, by allowing composition of smart contracts from different authors. We integrate a value system, named CSCoin, as a system smart contract to allow for accounting between those parties. However, the security model of Chainspace, is different from traditional unpermissioned blockchains, that rely on proof-of-work and global replication of state, such as Ethereum. In Chainspace smart contract authors designate the parts of the infrastructure that are trusted to maintain the integrity of their contract—and only depend on their correctness, as well as the correctness of contract sub-calls. This provides fine grained

control of which part of the infrastructure need to be trusted on a per-contract basis, and also allows for horizontal scalability.

### Contributions

This chapter makes the following key contributions:

- It presents Chainspace, a system that can scale arbitrarily as the number of nodes increase, tolerates Byzantine failures, and can be publicly audited.
- It introduces a distinction between parts of the smart contract that execute a computation, and those that check the computation and discusses how that distinction is key to supporting privacy-friendly smart-contracts.
- It presents a number of key system and application smart contracts and evaluates their performance. The contracts for privacy-friendly smart-metering and privacy-friendly polls illustrate and validate support for high-integrity and high-privacy applications.

### Outline

Section 3.1 presents an overview of Chainspace; Section 3.2 presents the client-facing application interface; Section 3.3 presents the design of internal data structures guaranteeing integrity, the distributed architecture, and smart contract definition and composition. Section 3.4 argues the correctness and security; specific smart contracts and their evaluations are presented in Section 3.5; Section 3.6 presents an evaluation of the smart contract performance; Section 3.7 presents limitation and Section 3.9 concludes the chapter.

## 3.1 Overview

Chainspace allows applications developers to implement distributed ledger applications by defining and calling procedures of smart contracts operating on controlled objects, and abstracts the details of how the ledger works and scales. In this section, we first describe data model of Chainspace, followed by an overview of the system design, its threat model and security properties.

### 3.1.1 Data Model: Objects, Contracts, Transactions.

Chainspace applies aggressively the end-to-end principle [130] in relying on untrusted end-user applications to build transactions to be checked and executed. We describe below key concepts within the Chainspace data model.

*Objects* are atoms that hold state in the Chainspace system. We usually refer to an object through the letter $o$, and a set of objects as $o \in O$. All objects have a cryptographically derived unique identifier used to unambiguously refer to the object, that we denote $\mathsf{id}(o)$. Objects also have a type, denoted as $\mathsf{type}(o)$, that determines the unique identifier of the smart contract that defines them, and a type name. In Chainspace object state is immutable. Objects may be in two meta-states, either *active* or *inactive*. Active objects are available to be operated on through smart contract procedures, while inactive ones are retained for the purposes of audit only. *Contracts* are special types of objects, that contain executable information on how other objects of types defined by the contract may be manipulated. They define a set of initial objects that are created when the contract is first created within Chainspace. A contract $c$ defines a *namespace* within which *types* (denoted as $\mathsf{types}(c)$) and a *checker v* for *procedures* (denoted as $\mathsf{proc}(c)$) are defined. A *procedure*, $p$, defines the logic by which a number of objects, that may be *inputs* or *references*, are processed by some logic and *local parameters* and *local return values* (denoted as $\mathsf{lpar}$ and $\mathsf{lret}$), to generate a number of object *outputs*. Notionally, input objects, denoted as a vector $\vec{w}$, represent state that is invalidated by the procedure; references, denoted as $\vec{r}$ represent state that is only read; and outputs are objects, or $\vec{x}$ are created by the procedure. Some of the local parameters or local returns may be secrets, and require confidentiality. We denote those as $\mathsf{spar}$ and $\mathsf{sret}$ respectively. We denote the execution of such a procedure as:

$$c.p(\vec{w},\vec{r},\mathsf{lpar},\mathsf{spar}) \rightarrow \vec{x},\mathsf{lret},\mathsf{sret} \tag{3.1}$$

for $\vec{w},\vec{r},\vec{x} \in O$ and $p \in \mathsf{proc}(c)$. We restrict the type of all objects (inputs $\vec{w}$, outputs $\vec{x}$ and references $\vec{r}$) to have types defined by the same contract $c$ as the procedure $p$ (formally: $\forall o \in \vec{w} \cup \vec{x} \cup \vec{r}.\mathsf{type}(o) \in \mathsf{types}(c)$). However, public locals (both $\mathsf{lpar}$ and

lret) may refer to objects that are from different contracts through their identifiers. We further require a procedure that outputs an non empty set of objects $\vec{x}$, to also take as parameters a non-empty set of input objects $\vec{w}$. Transactions that create no outputs are allowed to just take locals and references $\vec{r}$.

Associated with each smart contract $c$, we define a *checker* denoted as $v$. Those checkers are pure functions (ie. deterministic, and have no side-effects), and return a Boolean value. A checker $v$ is defined by a contract, and takes as parameters a procedure $p$, as well as inputs, outputs, references and locals.

$$c.v(p, \vec{w}, \vec{r}, \mathsf{lpar}, \vec{x}, \mathsf{lret}, \mathsf{dep}) \rightarrow \{\mathsf{true}, \mathsf{false}\} \tag{3.2}$$

Note that checkers do not take any secret local parameters (spar or sret). A checker for a smart contract returns true only if there exist some secret parameters spar or sret, such that an execution of the contract procedure $p$, with the parameters passed to the checker alongside spar or sret, is possible as defined in Equation (3.1). The variable dep represent the context in which the procedure is called: namely information about other procedure executions. This supports composition, as we discuss in detail in the next section. We note that procedures, unlike checkers, do not have to be pure functions, and may be randomized, keep state or have side effects. A smart contract defines explicitly the checker $c.v$, but does not have to define procedures *per se*. The Chainspace system is oblivious to procedures, and relies merely on checkers. Yet, applications may use procedures to create valid transactions. The distinction between procedures and checkers—that do not take secrets—is key to implementing privacy-friendly contracts.

*Transactions* represent the atomic application of one or more valid procedures to active input objects, and possibly some referenced objects, to create a number of new active output objects. The design of Chainspace is user-centric, in that a user client executes all the computations necessary to determine the outputs of one or more procedures forming a transaction, and provides enough evidence to the system to check the validity of the execution and the new objects. Once a transaction is accepted in the system it 'consumes' the input objects, that become inactive, and

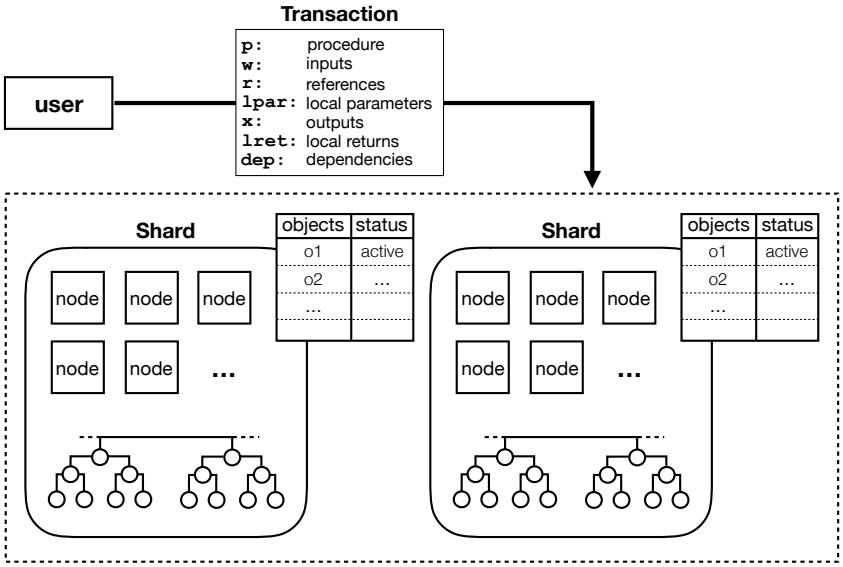


**Figure 3.2:** Design overview of Chainspace system, showing the interaction between users, transactions, objects and nodes in shards.

brings to life all new output objects that start their life by being active. References on the other hand must be active for the transaction to succeed, and remain active once a transaction has been successfully committed. A client packages enough information about the execution of those procedures to allow Chainspace to safely *serialize* its execution, and *atomically* commit it only if all transactions are valid according to relevant smart contract checkers.

### 3.1.2 System Design, Threat Model and Security Properties

We provide an overview of the system design, illustrated in Figure 3.2. Chainspace is comprised of a network of infrastructure *nodes* that manage valid objects, and ensure that only valid transactions are committed. A key design goal is to achieve scalability in terms of high transaction throughput and low latency. To this end, nodes are organized into shards that manage the state of objects, keep track of their validity, and record transactions aborted or committed. Within each shard all honest nodes ensure they consistently agree whether to accept or reject a transaction: whether an object is active or inactive at any point, and whether traces from contracts they know check. Across shards, nodes must ensure that transactions are *committed* if all shards are willing to commit the transaction, and rejected (or *aborted*) if any shards decide to abort the transaction—due to checkers returning false or objects

being inactive. To satisfy these requirements, Chainspace relies on a *cross-shard consensus protocol*. Consensus on committing (or aborting) transactions takes place in parallel across different shards. For transparency and auditability, nodes in each shard periodically publish a signed hash chain of *checkpoints*: shards add a block (Merkle tree) of evidence including transactions processed in the current epoch, and signed promises from other nodes, to the hash chain. Chainspace supports security properties against two distinct types of adversaries, both polynomial time bounded:

- **Honest Shards (HS).** The first adversary may create arbitrary contracts, and input arbitrary transactions into Chainspace, however they are bound to only control up to $f$ faulty nodes in any shard. As a result, and to ensure the correctness and liveness properties of Byzantine consensus, each shard must have a size of at least $3f + 1$ nodes.

- **Dishonest Shards (DS).** The second adversary has, additionally to HS, managed to gain control of one or more shards, meaning that they control over $f$ nodes in those shards. Thus, its correctness or liveness may not be guaranteed.

Faulty nodes in shards may behave arbitrarily, and collude to violate any of the security, safely or liveness properties of the system. They may emit incorrect or contradictory messages, as well as not respond to any or some requests. Given this threat model, Chainspace supports the following security properties:

- **Transparency.** Chainspace ensures that anyone in possession of the identity of a valid object may authenticate the full history of transactions and objects that led to the creation of the object. No transactions may be inserted, modified or deleted from that causal chain or tree. Objects may be used to self-authenticate its full history—this holds under both the HS and DS threat models.

- **Integrity.** Subject to the HS threat model, when one or more transactions are submitted only a set of valid non-conflicting transactions will be committed within the system. This includes resolving conflicts—in terms of multiple transactions using the same objects—ensuring the validity of the

transactions, and also making sure that all new objects are registered as active. Ultimately, Chainspace transactions are accepted, and the set of active objects changes, as if executed sequentially—however, unlike other systems such as Ethereum [24], this is merely an abstraction and high levels of concurrency are supported.

- **Encapsulation.** The smart contract checking system of Chainspace enforces strict isolation between smart contracts and their state—thus prohibiting one smart contract from directly interfering with objects from other contracts. Under both the HS and DS threat models. However, cross-contract calls are supported but mediated by well defined interfaces providing encapsulation.

- **Non-repudiation.** In case conflicting or otherwise invalid transactions were to be accepted in honest shards (in the case of the DS threat model), then evidence exists to pinpoint the parties or shards in the system that allowed the inconsistency to occur. Thus, failures outside the HS threat model, are detectable; the guilty parties may be banned; and appropriate off-line recovery mechanisms could be deployed.

## 3.2 The Chainspace Application Interface

Smart contract developers in Chainspace register a smart contract $c$ into the distributed system managing Chainspace, by defining a checker for the contract and some initial objects. Users may then submit transactions to operate on those objects in ways allowed by the checkers. Transactions represent the execution of one or more procedures from one or more smart contracts. It is necessary for all inputs to all procedures within the transaction to be active for a transaction to be executed and produce any output objects. Transactions are *atomic*: either all their procedures run, and produce outputs, or none of them do. Transactions are also *consistent*: in case two transactions are submitted to the system using the same active object inputs, at most one of them will eventually be executed to produce outputs. Other transactions, called *conflicting*, will be aborted.

$$\frac{\alpha_0,\ \mathit{Valid}(t),\alpha' \qquad \alpha',\ \mathit{Valid}(T'),\alpha_1}{\alpha_0,\ \mathit{Valid}(T = t :: T'),\alpha_1}\ \text{(Sequence)}$$

$$\frac{\alpha_0,\ \mathit{Valid}(\mathsf{dep}),\alpha' \qquad \alpha', c.v(p,\vec{w},\vec{r},\mathsf{lpar},\vec{x},\mathsf{lret},\mathsf{dep}),(\alpha' \setminus \vec{w}) \cup \vec{x} \qquad \begin{array}{c}\vec{w},\vec{r} \in \alpha' \wedge \\ (\vec{x} \neq \emptyset) \rightarrow (\vec{w} \neq \emptyset) \wedge \\ \forall o \in \vec{w} \cup \vec{x} \cup \vec{r}.\mathsf{type}(o) \in \mathsf{types}(c)\end{array}}{\alpha_0,\ \mathit{Valid}(t = [c,p,\vec{w},\vec{r},\vec{x},\mathsf{lpar},\mathsf{lret},\mathsf{dep}]),(\alpha' \setminus \vec{w}) \cup \vec{x}}\ \text{(Check)}$$

**Figure 3.3:** The sequencing and checking validity rules for transactions.

**Representation of transactions**. A transaction within Chainspace is represented by sequence of *traces* of the executions of the procedures that compose it, and their interdependencies. These are computed and packaged by end-user clients, and contain all the information a checker needs to establish its correctness. A Transaction is a data structure such that:

type *Transaction* : *Trace* list

type *Trace* : Record {

$c$ : id($o$), $p$ : string,

$\vec{w},\vec{r},\vec{x}$ : id($o$) list,

lpar, lret : arbitrary data,

dep : *Trace* list}

To generate a set of traces composing the transaction, a *user executes on the client side all the smart contract procedures* required on the input objects, references and local parameters, and generates the output objects and local returns for every procedure—potentially also using secret parameters and returns. Thus the actual computation behind the transactions is performed by the user, and the traces forming the transaction already contain the output objects and return parameters, and sufficient information to check their validity through smart contract checkers. This design pattern is related to traditional *optimistic concurrency control*.

Only valid transactions are eventually committed into the Chainspace system, as specified by two validity rules *sequencing* and *checking* presented in Figure 3.3. Transactions are considered valid within a context of a set of active objects maintained by Chainspace, denoted with $\alpha$. Valid transactions lead to a new context of active

objects (*e.g.* $\alpha'$). We denote this through the triplet $(\alpha, Valid(T), \alpha')$, which is true if the execution of transaction $T$ is valid within the context of active objects $\alpha$ and generates a new context of active objects $\alpha'$. The two rules are as follows:

- (Sequence rule). A '*Trace* list' (within a '*Transaction*' or list of dependencies) is valid if each of the traces are valid in sequence (see Figure 3.3 rule for sequencing). Further, the active objects set is updated in sequence before considering the validity of each trace.

- (Check rule). A particular '*Trace*' is valid, if the sequence of its dependencies are valid, and then in the resulting active object context, the checker for it returns `true`. A further three side conditions must hold: (i) inputs and references must be active; (ii) if the trace produces any output objects it must also contain some input objects; and (iii) all objects passed to the checker must be of types defined by the smart contract of this checker (see Figure 3.3 rule for checking).

The ordering of active object sets in the validation rules result in a depth-first validation of all traces, which represents a depth-first execution and data flow dependency between them. It is also noteworthy that only the active set of objects needs to be tracked to determine the validity of new transactions, which is in the order of magnitude of active objects in the system. The much longer list of inactive objects, which grows to encompass the full history of every object in the system is not needed—which we leverage to enable scaling when validating transactions. It also results in a smaller amount of working memory to perform incremental audits. A valid transaction is executed in a serialized manner, and committed or aborted atomically. If it is committed, the new set of active objects replaces the previous set; if not the set of active objects does not change. Determining whether a transaction may commit involves ensuring all the input objects are active, and all are consumed as a result of the transaction executing, as well as all new objects becoming available for processing (references however remain active). Chainspace ensures this through a distributed atomic commit protocol (Section 3.3.3).

**Smart contract composition**. A contract procedure may call a transaction of another smart contract, with specific parameters and rely upon returned values. This is achieved through passing the dep variable to a smart contract checker, a validated list of traces of all the sub-calls performed. The checker can ensure that the parameters and return values are as expected, and those dependencies are checked for validity by Chainspace. Composition of smart contracts is a key feature of a transparent and auditable computation platform. It allows the creation of a library of smart contracts that act as utilities for other higher-level contracts: for example, a simple contract can implement a cryptographic currency, and other contracts—for e-commerce for example—can use this currency as part of their logic. Section 6.4.1 shows how to take advantage of this feature to build a smart contract library allowing to issue and verify selective disclosure credentials. Furthermore, we compose smart contracts, in order to build some of the functionality of Chainspace itself as a set of 'system' smart contracts, including management of shards mapping to nodes, key management of shard nodes, and governance.

Chainspace also supports the atomic batch execution of multiple procedures for efficiency, that are not dependent on each other.

**Reads**. Besides executing transactions, Chainspace clients, need to read the state of objects, if anything, to correctly form transactions. Reads, by themselves, cannot lead to inconsistent state being accepted into the system, even if they are used as inputs or references to transactions. This is a result of the system checking the validity rules before accepting a transaction, which will reject any stale state. Thus, any mechanism may be used to expose the state of objects to clients, including traditional relational databases, or 'no-SQL' alternatives. Additionally, any indexing mechanism may be used to allow clients to retrieve objects with specific characteristics faster. Decentralized, read-only stores have been extensively studied, so we do not address the question of reads further in this work.

**Privacy by design**. Defining smart contract logic as checkers allows Chainspace to support privacy friendly-contracts by design. In such contracts some information in objects is not in the clear, but instead either encrypted using a public key, or

committed using a secure commitment scheme as [131]. The transaction only contains a valid proof that the logic or invariants of the smart contract procedure were applied correctly or hold respectively, and can take the form of a zero-knowledge proof, or a Succinct Argument of Knowledge (SNARK). Then, generalizing the approach of Miers *et al.* [42], the checker runs the verifier part of the proof or SNARK that validates the invariants of the transactions, without revealing the secrets within the objects to the verifiers.

## 3.3 The Chainspace System Design

In Chainspace a network of infrastructure *nodes* manages valid objects, and ensure key invariants: namely that only valid transactions are committed. We discuss the data structures nodes use collectively and locally to ensure high integrity; and the distributed protocols they employ to reach consensus on the accepted transactions.

### 3.3.1 High-Integrity Data Structures

Chainspace employs a number of high-integrity data structures. They enable those in possession of a valid object or its identifier to verify all operations that lead to its creation; they are also used to support *non-equivocation*—preventing Chainspace nodes from providing a split view of the state they hold without detection.

**Hash-DAG structure**. Objects and transactions naturally form a directed acyclic graph (DAG): given an initial state of active objects a number of transactions render their inputs invalid, and create a new set of outputs as active objects. These may be represented as a directed graph between objects, transactions and new objects and so on. Each object may only be created by a single transaction trace, thus cycles between future transactions and previous objects never occur. We prove that output object identifiers resulting from valid transactions are fresh (see Theorem 1). Hence, the graph of objects inputs, transactions and objects outputs form a DAG, that may be indexed by their identifiers. We leverage this DAG structure, and augment it to provide a high-integrity data structure. Our principal aim is to ensure that given an object, and its identifier, it is possible to unambiguously and unequivocally check all transactions and previous (now inactive) objects and references that contribute to the

existence of the object. To achieve this we define as an identifier for all objects and transactions a cryptographic hash that directly or indirectly depends on the identifiers of all state that contributed to the creation of the object.

Specifically, we define a function $\mathsf{id}(\mathit{Trace})$ as the identifier of a trace contained in transaction $T$. The identifier of a trace is a cryptographic hash function over the name of contract and the procedure producing the trace; as well as serialization of the input object identifiers, the reference object identifiers, and all local state of the transaction (but not the secret state of the procedures); the identifiers of the trace's dependencies are also included. Thus all information contributing to defining the Trace is included in the identifier, except the output object identifiers. We also define the $\mathsf{id}(o)$ as the identifier of an object $o$. We derive this identifier through the application of a cryptographic hash function, to the identifier of the trace that created the object $o$, as well as a unique name assigned by the procedures creating the trace, to this output object. (Unique in the context of the outputs of this procedure call, not globally, such as a local counter.) An object identifier $\mathsf{id}(o)$ is a high-integrity handle that may be used to authenticate the full history that led to the existence of the object $o$. Due to the collision resistance properties of secure cryptographic hash functions an adversary is not able to forge a past set of objects or transactions that leads to an object with the same identifier. Thus, given $\mathsf{id}(o)$ anyone can verify the authenticity of a trace that led to the existence of $o$. A very important property of object identifiers is that future transactions cannot re-create an object that has already become inactive. Thus checking object validity only requires maintaining a list of active objects, and not a list of past inactive objects.

**Security Theorem 1.** *No sequence of valid transactions, by a polynomial time constrained adversary, may re-create an object with the same identifier with an object that has already been active in the system.*

*Proof.* We argue this property by induction on the serialized application of valid transactions, and for each transaction by structural induction on the two validity rules. Assuming a history of $n-1$ transactions for which this property holds we consider transaction $n$. Within transaction $n$ we sequence all traces and their dependencies, and follow the data flow of

the creation of new objects by the 'check' rule. For two objects to have the same $\mathsf{id}(o)$ there need to be two invocations of the check rule with the same contract, procedure, inputs and references. However, this leads to a contradiction: once the first trace is checked and considered valid the active input objects are removed from the active set, and the second invocation becomes invalid. Thus, as long as object creation procedures have at least one input (which is ensured by the side condition) the theorem holds, unless an adversary can produce a hash collision. The inductive base case involves assuming that no initial objects start with the same identifier—which we can ensure axiomatically. □

We call this directed acyclic graph with identifiers derived using cryptographic functions a Hash-DAG, and we make extensive use of the identifiers of objects and their properties in Chainspace.

**Node hash-chains**. Each node in Chainspace, that is entrusted with preserving integrity, associates with its shard a hash chain. Periodically, peers within a shard consistently agree to seal a *checkpoint*, as a block of transactions into their hash chains. They each form a Merkle tree containing all transactions that have been accepted or rejected in sequence by the shard since the last checkpoint was sealed. Then, they extend their hash chain by hashing the root of this Merkle tree and a block sequence number, with the head hash of the chain so far, to create the new head of the hash chain. Each peer signs the new head of their chain, and shares it with all other peers in the shard, and anyone who requests it. For strong auditability additional information, besides committed or aborted transactions, has to be included in the Merkle tree: node should log any promise to either commit or abort a transaction from any other peer in any shard (see Section 3.3.3). All honest nodes within a shard independently create the same chain for a checkpoint, and a signature on it—as long as the consensus protocols within the shards are correct. We say that a checkpoint represents the decision of a shard, for a specific sequence number, if at least $f+1$ signatures of shard nodes sign it. On the basis of these hash chains we define a *partial audit* and a *full audit* of the Chainspace system.

In a *partial audit* a client is provided evidence that a transaction has been either committed or aborted by a shard. A client performing the partial audit may request

from any node of the shard evidence for a transaction T. The shard peer will present a block representing the decision of the shard, with $f+1$ signatures, and a proof of inclusion of a commit or abort for the transaction, or a signed statement the transaction is unknown. A partial audit provides evidence to a client of the fate of their transaction, and may be used to detect past of future violations of integrity. A partial audit is an efficient operation since the evidence has size $O(s+\log N)$ in $N$ the number of transactions in the checkpoint and $s$ the size of the shard—thanks to the efficiency of proving inclusion in a Merkle tree, and checking signatures.

A *full audit* involves replaying all transactions processed by the shard, and ensuring that (i) all transactions were valid according to the checkers the shard executed; (ii) the objects input or references of all committed transactions were all active (see rules in Figure 3.3); and (iii) the evidence received from other shards supports committing or aborting the transactions. To do so an auditor downloads the full hash-chain representing the decisions of the shard from the beginning of time, and re-executes all the transactions in sequence. This is possible, since—besides their secret signing keys—peers in shards have no secrets, and their execution is deterministic once the sequence of transactions is defined. Thus, an auditor can re-execute all transactions in sequence, and check that their decision to commit or abort them is consistent with the decision of the shard. Doing this, requires any inter-shard communication (namely the promises from other shards to commit or abort transactions) to be logged in the hash-chain, and used by the auditor to guide the re-execution of the transactions. A full audit needs to re-execute all transactions and requires evidence of size $O(N)$ in the number $N$ of transactions. This is costly, but may be done incrementally as new blocks of shard decisions are created.

### 3.3.2 Distributed Architecture & Consensus

A network of *nodes* manages the state of Chainspace objects, keeps track of their validity, and record transactions that are seen or that are accepted as being committed.

Chainspace uses sharding strategies to ensure scalability: a public function $\phi(o)$ maps each object $o$ to a set of nodes, we call a *shard*. These nodes collectively are entrusted to manage the state of the object, keep track of its validity, record

transactions that involve the object, and eventually commit at most one transaction consuming the object as input and rendering it inactive. However, nodes must only record such a transaction as committed if they have certainty that all other nodes have, or will in the future, record the same transaction as consuming the object. We call this distributed algorithm the *consensus* algorithm within the shard. For a transaction $T$ we define a set of *concerned nodes*, $\Phi(T)$ for a transaction structure $T$. We first denote as $\zeta$ the set of all objects identifiers that are input into or referenced by any trace contained in $T$. We also denote as $\xi$ the set of all objects that are output by any trace in $T$. The function $\Phi(T)$ represents the set of nodes that are managing objects that should exist, and be active, in the system for $T$ to succeed. More mathematically, $\Phi(T) = \bigcup\{\phi(o_i) | o_i \in \zeta \setminus \xi\}$, where $\zeta \setminus \xi$ represents the set of objects input but not output by the transaction itself (its free variables). The set of concerned peers thus includes all shard nodes managing objects that already exist in Chainspace that the transaction uses as references or inputs.

An important property of this set of nodes holds, that ensures that all smart contracts involved in a transaction will be mapped to some concerned nodes that manage state from this contract:

**Security Theorem 2.** *If a contract c appears in any trace within a transaction $T$, then the concerned nodes set $\Phi(T)$ will contain nodes in a shard managing an object o of a type from contract c. I.e. $\exists o.\mathsf{type}(o) \in \mathsf{types}(c) \wedge \phi(o) \cap \Phi(T) \neq \emptyset$.*

*Proof.* Consider any trace $t$ within $T$, from contract $c$. If the inputs or references to this trace are not in $\xi$—the set of objects that were created within $T$—then their shards will be included within $\Phi(T)$. Since those are of types within $c$ the theorem holds. If on the other hand the inputs or references are in $\xi$, it means that there exists another trace within $T$ from the same contract $c$ that generated those outputs. We then recursively apply the case above to this trace from the same $c$. The process will terminate with some objects of types in $c$ and shard managing them within the concerned nodes set—and this is guarantee to terminate due to the Hash-DAG structure of the transactions (that may have no loops). □

Security Theorem 2 ensures that the set of concerned nodes, includes nodes that manage objects from all contracts represented in a transaction. Chainspace leverages

this to distribute the process of rule validation across peers in two ways:

- For any existing object $o$ in the system, used as a reference or input within a transaction $T$, only the shard nodes managing it, namely in $\phi(o)$, need to check that it is active (as part of the 'check' rule in Figure 3.3).

- For any trace $t$ from contract $c$ within a transaction $T$, only shards of concerned nodes that manage objects of types within $c$ need to run the checker of that contract to validate the trace (again as part of the 'check' rule), and that all input, output and reference objects are of types within $c$.

However, all shards containing concerned nodes for $T$ need to ensure that all others have performed the necessary checks before committing the transaction, and creating new objects. There are many options for ensuring that concerned nodes in each shards do not reach an inconsistent state for the accepted transactions, such as Nakamoto consensus through proof-of-work [17], two-phase commit protocols, and classical consensus protocols like Paxos [132], PBFT [15], or xPaxos [87]. However, these approaches lack in performance, scalability, and/or security. Chainspace requires a scalable and decentralized mechanism to achieve consensus across shards.

### 3.3.3 Cross-Shard Consensus Protocol

Chainspace entrusts each object to a shard of nodes, that keep track of whether it exists, it is active or inactive. Within each shard all honest nodes must ensure they consistently agree whether to accept or reject a transaction: whether an object is active or inactive at any point, and whether traces from contracts they know check. Across shards, nodes must ensure that transactions are *committed* if all shards are willing to commit the transaction, and rejected (or *aborted*) if any shards decide to abort the transaction—due to it being invalid or objects involved being inactive. Chainspace requires a sharded consensus protocol for transaction processing in the *Byzantine* and *asynchronous* setting. Such protocols are delicate to build, and are the subject of Chapter 4. For the rest of this chapter, we assume Chainspace implements a cross-shard consensus protocol providing liveness, consistency and validity (see Section 3.4), and treats it as a black box.

Each transaction $T$ involves a fixed number of *concerned nodes* $\Phi(T)$ within Chainspace, corresponding to the shards managing its inputs and references. If two transactions $T_0$ and $T_1$ have disjoint sets of concerned nodes ($\Phi(T_0) \cap \Phi(T_1) = \emptyset$) they cannot conflict, and are executed in parallel or in any arbitrary order. Even if this set is not empty, but instead the input objects of those two transactions are disjoint, this property holds. If however, two transactions have common input objects, only one of them is accepted by all nodes. This is achieved through the cross-shard consensus protocol. It is local, in that it concerns only nodes managing the conflicting transactions, and does not require a global consensus. From the point of view of scalability, this protocol allows Chainspace's capacity grows linearly as more shards are added, subject to transactions having on average a constant, or sub-linear, number of inputs and references. Furthermore, those inputs must be managed by different nodes within the system to ensure that load of accepting transactions is distributed across them. How such distribution is achieved depends on the threat model of the application. Some may opt for a totally peer to peer model, where an ad-hoc random quorum of peers manages each object. Other application may opt for a small set of well-known authorities managing peers, with each object being managed by at least one representative peer from each authority.

## 3.4 Security and Correctness

Chainspace relies on a black box cross-shard consensus protocol (see Chapter 4), on which rest the security of Chainspace, namely *liveness*, *consistency*, and *validity*. Those properties hold under the 'honest shards' threat model (see Section 3.1.2). Liveness ensures that transactions make progress, and no locks are held indefinitely on objects, preventing other transactions from making progress. Consistency ensures that the execution of valid transactions could be serialized, and thus is correct. Validity ensures a transaction may only be committed if it is valid according to the smart contract checkers matching the traces of the procedures it executes.

### 3.4.1 Auditability

In the previous sections we show that if each shard contains at most $f$ faulty nodes (honest shard model), the cross-shard consensus protocol guarantees consistency and validity. In this section we argue that if this assumption is violated, *i.e.* one or more shards contain more than $f$ byzantine nodes each, then honest shards can detect faulty shards. Namely, enough auditing information is maintained by honest nodes in Chainspace to detect inconsistencies and attribute them to specific shards (or nodes within them). The rules for transaction validity are summarized in Figure 3.3. Those rules are checked in a distributed manner: each shard keeps and checks the active or inactive state of objects assigned to it; and also only the contract checkers corresponding to the type of those objects. An honest shard emits a pre-accept($T$) for a transaction T only if those checks pass, and pre-abort($T$) otherwise or if there is a lock on a relevant object. A dishonest shard may emit any of those messages arbitrarily without checking the validity rules. By definition, an invalid transaction is one that does not pass one or more of the checks defined in Figure 3.3 at a shard, for which the shard has erroneously emitted a pre-accept($T$) or pre-abort($T$) message.

**Security Theorem 3** (Auditability)**.** *A malicious shard (with more than f faulty nodes) that attempts to introduce an invalid transaction or object into the state of one or more honest shards, can be detected by an auditor performing a full audit of the Chainspace system.*

*Proof.* We consider two hash-chains from two distinct shards. We define the pair of them as being valid if (i) they are each valid under full audit, meaning that a re-execution of all their transactions under the messages received yields the same decisions to commit or abort all transactions; and (ii) if all pre-accept($T$) and pre-abort($T$) messages in one chain are compatible with all messages seen in the other chain. In this context 'compatible' means that all pre-accept($T$) and pre-abort($T$) statements received in one shard from the other represent the 'correct' decision to commit or abort the transaction T in the other shard. An example of incompatible message would result in observing a pre-accept($T$) message being emitted from the first shard to the second, when in fact the first shard should have aborted the transaction, due to the checker showing it is invalid or an input being inactive.

Due to the property of digital signatures (unforgeability and non-repudiation), if two hash-chains are found to be 'incompatible', one belonging to an honest shard and one belonging to a dishonest shard, it is possible for everyone to determine which shard is the dishonest one. To do so it suffices to isolate all statements that are signed by each shard (or a peer in the shard)—all of which should be self-consistent. It is then possible to show that within those statements there is an inconsistency—unambiguously implicating one of the two shards in the cheating. Thus, given two hash-chains it is possible to either establish their consistency, under a full audit, or determine which belongs to a malicious shard. □

Note that the mechanism underlying tracing dishonest shards is an instance of the age-old double-entry book keeping[1]: shards keep records of their operations as a non-repudiable signed hash-chain of checkpoints—with a view to prove the correctness of their operations. They also provide non-repudiable statements about their decisions in the form of signed pre-accept($T$) and pre-abort($T$) statements to other shards. The two forms of evidence must be both correct and consistent—otherwise their misbehaviour is detected.

## 3.5 System and Applications Smart Contracts

We present a number of key system and application smart contracts; the contracts for privacy-friendly smart-metering and privacy-friendly polls illustrate and validate support for high-integrity and high-privacy applications.

### 3.5.1 System Contracts

The operation of a Chainspace distributed ledger itself requires the maintenance of a number of high-integrity high-availability data structures. Instead of employing an ad-hoc mechanism, Chainspace employs a number of *system smart contracts* to implement those. Effectively, an instantiation of Chainspace is the combination of nodes running a consensus protocol, as well as a set of system smart contracts providing flexible policies about managing shards, smart contract creation, auditing and accounting. This section provides an overview of system smart contracts.

**Shard management**. The discussion of Chainspace so far, has assumed a function

[1] The first reported use is 1340AD [133].

$\phi(o)$ mapping an object $o$ to nodes forming a shard. However, how those shards are constituted has been abstracted. A smart contract ManageShards is responsible for mapping nodes to shards. ManageShards initializes a singleton object of type MS.Token and provides three procedures: MS.create takes as input a singleton object, and a list of node descriptors (names, network addresses and public verification keys), and creates a new singleton object and a MS.Shard object representing a new shard; MS.update takes an existing shard object, a new list of nodes, and $2f + 1$ signatures from nodes in the shard, and creates a new shard object representing the updated shard. Finally, the MS.object procedure takes a shard object, and a non-repudiable record of malpractice from one of the nodes in the shard, and creates a new shard object omitting the malicious shard node—after validating the misbehaviour. Note that Chainspace is 'open' in the sense that any nodes may form a shard; and anyone may object to a malicious node and exclude it from a shard.

**Smart-contract management**. Chainspace is also 'open' in the sense that anyone may create a new smart contract, and this process is implemented using the ManageContracts smart contract. ManageContracts implements three types: MC.Token, MC.Mapping and MC.Contract. It also implements at least one procedure, MC.create that takes a binary representing a checker for the contract, an initialization procedure name that creates initial objects for the contract, and the singleton token object. It then creates a number of outputs: one object of type MC.Token for use to create further contracts; an object of type MC.Contract representing the contract, and containing the checker code, and a mapping object MC.mapping encoding the mapping between objects of the contract and shards within the system. Furthermore, the procedure MC.create calls the initialization function of the contract, with the contract itself as reference, and the singleton token, and creates the initial objects for the contract. Note that this simple implementation for ManageContracts does not allow for updating contracts. The semantics of such an update are delicate, particularly in relation to governance and backwards compatibility with existing objects. The definitions of more complex, but correct, contracts for managing contracts are left for future work.

**Payments for processing transactions**. Chainspace is an open system, and requires protection againt abuse resulting from overuse. To achieve this we implement a method for tracking value through a contract called CSCoin. The CSCoin contract creates a fixed initial supply of coins—a set of objects of type The CSCoin.Account that may only be accessed by a user producing a signature verified by a public key denoted in the object. A CSCoin.transfer procedure allows a user to input a number of accounts, and transfer value between them, by producing the appropriate signature from incoming accounts. It produces a new version of each account object with updated balances. This contract has been implemented in Python with approximately 200 lines of code. The CSCoin contract is designed to be composed with other procedures, to enable payments for processing transactions. The transfer procedure outputs a number of local returns with information about the value flows, that may be used in calling contracts to perform actions conditionally on those flows. Shards may advertise that they will only consider actions valid if some value of CSCoin is transferred to their constituent nodes. This may apply to system contracts and application contracts.

### 3.5.2 Application Level Smart Contracts

This section describes some examples of privacy-friendly smart contracts and showcases how smart contract creators may use Chainspace to implement advanced privacy mechanisms.

**Sensor—'Hello World' contract**. To illustrate Chainspace's applications, we implement a simple 150 lines contract aggregating data from different sensors, called Sensor. This contract defines the types Sensor.Token and Sensor.Data, and two procedures Sensor.createSensor and Sensor.addData. The procedure Sensor.createSensor takes as input the singletone token (that is created upon contract creation), and outputs a fresh Sensor.Data object with initially no date. The Sensor.addData procedure is applied on a Sensor.Data object with some new sensor's data as parameter; it then creates a new object Sensor.Data appending the list of new data to the previous ones.

**Smart-Meter private billing**. A body of work [134] examines how to achieve

privacy-friendly time of use billing for smart meter deployments—a use-case requiring both high-integrity, and also privacy. Thus it showcases how smart contract creators may use Chainspace to implement advanced privacy mechanisms.

We implement a basic private smart-meter billing mechanism [135, 134] using the contract SMet: it implements three types SMet.Token, SMet.Meter and SMet.Bill; and three procedures, SMet.createMeter, SMet.AddReading, and SMet.computeBill. The procedure SMet.createMeter takes as input the singletone token and a public key and signature as local parameters, and it outputs a SMet.Meter object tied to this meter public key if the signature matches. SMet.Meter objects represent a collection of readings and some meta-data about the meter. Subsequently, the meter may invoke SMet.addReading on a SMet.Meter with a set of cryptographic commitments readings and a period identifier as local parameters, and a valid signature on them. A signature is also included and checked to ensure authenticity from the meter. A new object SMet.Meter is output appending the list of new readings to the previous ones. Finally, a procedure SMet.computeBill is invoked with a SMet.Meter and local parameters a period identifier, a set of tariffs for each reading in the period, and a zero-knowledge proof of correctness of the bill computation. The procedure outputs a SMet.Bill object, representing the final bill in plain text and the meter and period information. This proof of correctness is provided to the checker—rather than the secret readings—which proves that the readings matching the available commitments and the tariffs provided yield the bill object. The role of the checker, which checks public data, in both those cases is very different from the role of the procedure that is passed secrets not available to the checkers to protect privacy. This contract is implemented in about 200 lines of Python and is evaluated in section Section 3.6.

**A Platform for decision making**. An additional example of Chainspace's privacy-friendly application is a smart voting system. We implement the contract SVote with three types, SVote.Token, SVote.Vote and SVote.Tally; and three procedures. SVote.createElection, consumes a singleton token and takes as local parameters the options, a list of all voter's public key, the tally's public key, and a signature on them from the tally. It outputs a fresh SVote.Vote object, representing the initial stage

of the election (all candidates having a score of zero) along with a zero-knowledge proof asserting the correctness of the initial stage. SVote.addVote, is called on a SVote.Vote object and takes as local parameters a new vote to add, homomorphically encrypted and signed by the voter. In addition, the voter provides a zero-knowledge proof certifying that her vote is a binary value and that she voted for exactly one option. The voter's public key is then removed from the list of participants to ensure that she cannot vote more than once. If all proofs are verified by the checker and the voter's public key appears in the list, a new SVote.Vote object is created as the homomorphic addition of the previous votes with the new one. Note that the checker does not need to know the clear value of the votes to assert their correctness since it only has to verify the associated signatures and zero-knowledge proofs. Finally, the procedure SVote.tally is called to threshold decrypt the aggregated votes and provide a SVote.Tally object representing the final election's result in plain text, along with a proof of correct decryption from the tally. The SVote contract's size is approximately 400 lines of Python and is also evaluated in Section 3.6.

## 3.6 Smart Contract Evaluation

We build a Python contracts environment allowing developers to write, deploy and test smart contracts. These are deployed on each node by running the Python script for the contract, which starts a local web service for the contract's checker. The contract's checker is then called though the web service. The environment provides a framework to allow developers to write smart contracts with little worry about the underlying implementation, and provides an auto-generated checker for simple contracts. We are releasing the code as an open-source project[2]. We evaluate the cost and performance of some smart contracts described in Section 3.5.1. We compute the mean and standard deviation of the execution of each procedure (denoted as [g]) and checker (denoted as [c]) in the contracts. Each figure is the result of 10,000 measured on a dual-core Apple MacBook Pro 4.1, 2.7GHz Intel Core i7. The last column indicates the transaction's size resulting from executing the procedure.

[2] https://github.com/chainspace/chainspace

| Sensor—Contract size: ∼150 lines | | | | |
|---|---|---|---|---|
| **Operation** | | **Mean (ms)** | **Std. (ms)** | **Size (B)** |
| createSensor | [g] | 0.139 | ± 0.039 | 416 |
| | [c] | 0.021 | ± 0.008 | - |
| addData | [g] | 0.105 | ± 0.085 | 417 |
| | [c] | 0.036 | ± 0.015 | - |

**Table 3.1:** Chainspace Sensor contract.

| CSCoin—Contract size: ∼200 lines | | | | |
|---|---|---|---|---|
| **Operation** | | **Mean (ms)** | **Std. (ms)** | **Size (B)** |
| createAccount | [g] | 4.845 | ± 0.683 | 512 |
| | [c] | 0.022 | ± 0.005 | - |
| authTransfer | [g] | 4.986 | ± 0.684 | 1114 |
| | [c] | 5.750 | ± 0.474 | - |

**Table 3.2:** Chainspace CSCoin contract.

All cryptographic operations as digital signatures and zero-knowledge proofs are implemented using the Python library petlib [136], wrapping OpenSSL.

The Sensor contract (Table 3.1) is very cheap since it does not contain any cryptographic operation; the checker only has to verify the format of the transaction. The resulting micro benchmarks for createSensor is therefore a good indication of just the overhead of the Chainspace system—to execute a procedure or a checker. The user needs to generate a signing key pair to create an account in the CSCoin contract, which takes about 5 ms. However, verifying the account creation only requires to check the transaction's format, and it is therefore very fast. Transferring money is a little more expensive due to the need to sign the amount transferred and the beneficiary, and verifying the signature in the checker.

Similarly to CSCoin (Table 3.2), creating a meter (Table 3.3) requires generating a cryptographic key pair which takes about 5 ms, while verifying the meter's creation is faster and only requires checking the transaction's format. Adding new readings takes about 5 ms, as the user needs to create a signed commitment of the readings which requires elliptic curve operations and an ECDSA signature. Computing the bill takes slightly longer (5.8 ms), and involves homomorphic additions, and verifying the bill involves checking a zero-knowledge proof of the billing calculation.

| SMet—Contract size: ∼200 lines | | | | |
|---|---|---|---|---|
| **Operation** | | **Mean (ms)** | **Std. (ms)** | **Size (B)** |
| createMeter | [g] | 4.786 | ± 0.480 | ∼600 |
| | [c] | 0.060 | ± 0.003 | - |
| addReading | [g] | 5.286 | ± 0.506 | ∼1100 |
| | [c] | 5.965 | ± 0.697 | - |
| computeBill | [g] | 5.043 | ± 0.513 | ∼1100 |
| | [c] | 5.870 | ± 0.603 | - |

**Table 3.3:** Chainspace SMet contract.

| SVote—Contract size: ∼400 lines | | | | |
|---|---|---|---|---|
| **Operation** | | **Mean (ms)** | **Std. (ms)** | **Size (B)** |
| createElection | [g] | 11.733 | ± 1.028 | ∼1227 |
| | [c] | 11.327 | ± 0.782 | - |
| addVote | [g] | 14.086 | ± 1.043 | ∼2758 |
| | [c] | 28.178 | ± 1.433 | - |
| tally | [g] | 253.286 | ± 7.793 | ∼1264 |
| | [c] | 11.589 | ± 0.937 | - |

**Table 3.4:** Chainspace SVote contract.

The SVote contract (Table 3.4) is more expensive than the others since it extensively uses zero-knowledge proofs and more advanced cryptography. For simplicity, this smart contract is tested with three voters and two options. First, creating a new election event requires building a signed homomorphic encryption of the initial value for each option, and a zero-knowledge proof asserting that the encrypted value is zero; this takes roughly 11 ms to generate the transaction and to run the checker. Next, each time a vote is added, the user proves two zero-knowledge statements—one asserting that she votes for exactly one option and one proving that her vote is a binary value—and computes an ECDSA signature on her vote, which takes about 11 ms and generates a transaction of about 2.7 kB. Verifying the signature and the two zero-knowledge proofs are slower and takes about 30 ms. Finally, tallying is the slowest operation since it requires to threshold decrypt the homomorphic encryption of the votes' sum.

## 3.7 Limitations

Chainspace has a number of limitations, that are beyond the scope of this work to tackle, and deferred to future work. The integrity properties of Chainspace rely on all shards managing objects being honest, namely containing at most $f$ fault nodes each. We allow any set of nodes to create a shard. However, this means that the function $\phi(o)$ mapping objects to shards must avoid dishonest shards. Our isolation properties ensure that a dishonest shard can at worse affect state from contracts that have objects mapped to it. Thus, in Chainspace, we opt to allow the contract creator to designate which shards manage objects from their contract. This embodies specific trust assumptions where users have to trust the contract creator both for the code (which is auditable) and also for the choice of shards to involve in transactions—which is also public. In case one or more shards are malicious, we provide an auditing mechanism for honest nodes in honest shards to detect the inconsistency and to trace the malicious shard. Through the Hash-DAG structure it is also possible to fully audit the histories of two objects, and to ensure that the validity rules hold jointly—in particular the double-use rules. However, it is not clear how to automatically recover from detecting such an inconsistency. Options include: forcing a fork into one or many consistent worlds; applying a rule to collectively agree the canonical version; patching past transactions to recover consistency; or agree on a minimal common consistent state. Which of those options is viable or best is left as future work. Checkers involved in validating transactions can be costly. For this reason we allow peers in a shard to accept transactions subject to a SCCoin payment to the peers. However, this 'flat' fee is not dependent on the cost or complexity of running the checker which might be more or less expensive. Ethereum [24] instead charges 'gas' according to the cost of executing the contract procedure—at the cost of implementing their own virtual machine and language. Finally, we present in Section 3.5.1 'payments for processing transactions' to address DoS through transaction fees. However, this mechanism only offers a partial solutions, since operations with higher costs to the system would need to cost more, and all infrastructure nodes would have to be proper incentives to continue participating in

the protocols. A through study of such mechanisms is left for future work.

## 3.8 Comparison with Related Works

We present some recent systems that provide a transparent platform based on blockchains for smart contracts, and compare it with Chainspace.

**Omniledger**. The most comparable system to Chainspace is Omniledger [4]—that was developed concurrently—and provides a scalable distributed ledger for a cryptocurrency, and cannot support generic smart contracts (in contrast with Chainspace which is a smart contract platform). Omniledger assigns nodes (selected using a Sybil-attack resistant mechanism) into shards among which state, representing coins, is split. The node-to-shard assignment is done every epoch using a bias-resistant decentralized randomness protocol [104] to prevent an adversary from compromising individual shards. Similarly to Chainspace, a block-DAG (Directed Acyclic Graph) structure is maintained in each shard rather than a single blockchain, effectively creating multiple blockchains in which consensus of transactions can take place in parallel. Nodes within shards reach consensus through the PBFT protocol [15] with ByzCoin [95]'s modifications that enable $O(n)$ messaging complexity. In contrast, Chainspace relies on a black box consensus protocols that can be implemented with any PBFT variant without breaking any security assumptions.

**Ethereum**. Ethereum [24] provides a decentralized virtual machine, called EVM, for executing smart-contracts. Its main scalability limitation results from every node having to process every transaction, as Bitcoin. On the other hand, Chainspace's sharded architecture allows for a ledger linearly scalable since only the nodes concerned by the transaction—that is, managing the transaction's inputs or references—have to process it. Ethereum plans to improve scalability through sharding techniques [137], but their work is still theoretical and is not implemented or measured. One major difference with Chainspace is that Ethereum's smart contract are executed by the node, contrarily to the user providing the outputs of each transaction. Chainspace also supports smart contracts written in any kind of language as long as checkers are pure functions, and there are no limitations for the code creating transactions. Some

industrial systems [138, 139] implement similar functionalities as Chainspace, but with little empirical performance evaluation. In terms of security policy, Chainspace system implements a platform that enforces high-integrity by embodying a variant of the Clark-Wilson [140], proposed before smart contracts were heard of.

**Tezos**. Tezos [141] has a strong type checking system and implements its cryptocurrency as an 'account smart contract. However, Tezos's smart contracts are statefull and updating a balance requires to rewrite the contract's storage space. This introduces many complications to prevent replay attacks and transaction's validity's check. Chainspace avoid this situation by producing new version of account objects as specified in the Security Theorem 1 (see Section 3.4). Moreover, Tezos implements an Ethereum-like gas system to pay nodes to execute the smart contract.

## 3.9 Chapter Summary

Chainspace is an open, distributed ledger platform for high-integrity and transparent processing of transactions. Chainspace offers extensibility though privacy-friendly smart contracts. We presented an instantiation of Chainspace by parameterizing it with a number of 'system' and 'application' contracts, along with their evaluation. However, unlike existing smart-contract based systems such as Ethereum [24], it offers high scalability through sharding across nodes, while offering high auditability. As such it offers a competitive alternative to both centralized and permissioned systems, as well as fully peer-to-peer, but unscalable systems like Ethereum.

## Chapter 4

# Replay Attacks and Defenses Against Cross-shard Consensus

Chapter 3 describes Chainspace assuming it implements a black box cross-shard consensus protocol; this chapter opens that black box. We first present a family of replay attacks against existing cross-shard consensus protocols, illustrating that designing such protocols is a delicate task. We then describe the issues that lead to these vulnerabilities, and present Byzcuit, a novel cross-shard consensus protocol that is immune to those attacks. Figure 4.1 highlights Byzcuit, the cross-shard consensus protocol run at the heart of Chainspace.

As explained in the previous chapters, the key idea is to create groups (or shards) of nodes that handle only a subset of all transactions and system state, relying on classical Byzantine Fault Tolerance (BFT) protocols for reaching *intra-shard consensus*. These systems achieve optimal performance and scalability because: (i) non-conflicting transactions can be processed in parallel by multiple shards; and (ii) the system can scale up by adding new shards. However, this separation of transaction handling across shards is not perfectly 'clean'—a transaction might rely on data managed by multiple shards, requiring an additional step of *cross-shard consensus* across the concerned shards (see arrows on Figure 4.1). An atomic commit protocol (such as the two-phase commit protocol [52]) typically runs across all the concerned shards to ensure the transaction is accepted by all or none of those shards.

**Vulnerabilities in previous systems**. This chapter presents the first replay attacks on

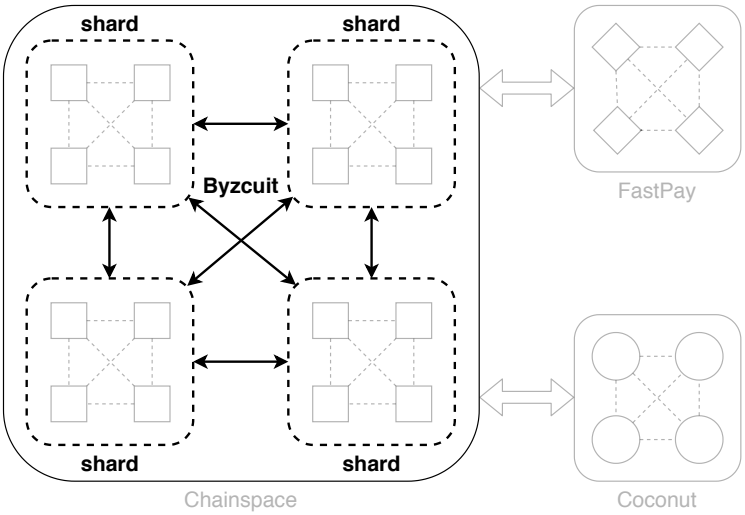


**Figure 4.1:** Global overview: Byzcuit.

cross-shard consensus in sharded protocols. An attacker can launch these attacks with minimal effort, without subverting any nodes, and assuming a weakly synchronous network (and in some cases, without relying on any network assumption)—even when the byzantine safety assumptions are satisfied. These attacks compromise key system properties of safety and liveness, effectively enabling the attacker to double-spend coins (or any other objects managed by the blockchain) and create coins out of thin air. We concretely sketch the replay attacks in the context of two representative systems: S-BAC [25] and Atomix [4] . Section 4.4 describes how an attacker can actively stage the attack by eliciting from the system the messages to replay (in contrast to passively observing the network traffic, and waiting to detect and record the target messages). We also discuss the feasibility of these attacks and their real-world impact. The replay attacks we present are generic and apply to other systems that are based on similar models, like RapidChain [5]. Based on our detailed analysis of replay attacks, we develop a defense strategy (Section 4.5).

**Byzcuit**. Drawing insights from our analysis of performance trade-offs and replay attack vulnerabilities in existing cross-shard consensus protocols, we present a hybrid system, Byzcuit (Section 4.6). It combines useful features from previous designs to achieve high performance and scalability, and leverages our proposed defense to achieve resilience against replay attacks. Byzcuit employs a Transaction Manager to

coordinate cross-shard communication, reducing its cost to $O(n)$ communication, amongst $n$ shards, in the absence of faults. We implement a prototype of Byzcuit, and release it as an open-source project[1]. We evaluate Byzcuit on a real cloud-based testbed under varying transaction loads and show that Byzcuit has a client-perceived latency of less than a second, even for a system load of 1,000 transactions per second (tps). Byzcuit's transaction throughput scales linearly with the number of shards by 250–300 tps for each shard added, handling up to 1,550 tps with 10 shards. We quantify the overhead of our replay defenses and find that as expected those reduce the throughput by 20–250 tps.

## Contributions

This chapter makes the following key contributions:

- It develops the first replay attacks against cross-shard consensus protocols, and illustrate their impact on important academic and implemented designs
- It presents defenses against those replay attacks, and discusses the issues that lead to these vulnerabilities.
- It designs a hybrid, new system Byzcuit with improved performance trade-offs, and which integrates our proposed defense to achieve resilience against the replay attacks;
- It implements a prototype of Byzcuit and evaluate its performance and scalability on a real distributed set of nodes and under varying transaction loads, and illustrate how it is superior to previous proposals.

## Outline

Section 4.1 presents an overview of the replay attacks; Section 4.2 describes replay attacks on shard-led cross-shard consensus protocols; and Section 4.3 describe replay attacks on client-led cross-shard consensus protocols. Section 4.5 discusses the issues that lead to the replay attacks describes how to fix them; section 4.6 presents

[1] `https://github.com/sheharbano/byzcuit`

Byzcuit, a system achieving resilience against the replay attacks; Section 4.7 presents an evaluation of Byzcuit; and Section 4.8 concludes the chapter.

## 4.1 Attack Overview

Sections 4.2 and 4.3 discuss replay attacks on both shard-led and client-led cross-shard consensus protocols, respectively. We provide a high-level description of these attacks and the threat model, and describe the notation we use.

**Replay attacks on cross-shard consensus**. The attacker records a target shard's responses to the atomic commit protocol, and replays them during another instance of the protocol. We present two families of replay attacks: (i) attacks against the first phase (*voting*), and (ii) attacks against the second phase (*commit*) of the atomic commit protocol. To attack the first phase (*voting*) of the atomic commit protocol, the attacker replaces messages generated by the target shard by replaying prerecorded messages. In practice, the attacker does not *replace* those messages—it achieves a similar result by making its replayed messages arrive at the coordinator faster (racing the target shard's original message), exploiting the fact that the coordinator makes progress based on the first message it receives. Replaying messages in this fashion enables the attacker to compromise the system safety (by creating inconsistent state on the shards) and/or liveness (by causing valid transactions to be rejected). To attack the second phase (*commit*) of the atomic commit protocol, the attacker simply replays prerecorded messages to target shards, and compromises consistency. The attacker can replay those messages at any time of its choice, and does not rely on any racing condition as in the previous case.

**Threat model**. The attacker can successfully launch the described attacks without colluding with any shard nodes, and under the BFT honest majority safety assumption for nodes within shards (*i.e.*, the attacks are effective even if *all* nodes are honest). We assume an attacker that can observe and record messages generated by shards; this can be achieved by (i) monitoring the network, or (ii) reading the blockchain (which is more practical). The attacker can be an external observer that passively collects the target messages at the level of the network, or it can act as a client and

actively interact with the system to elicit the target messages. The attacks against the first phase of the atomic commit protocol (Sections 4.2.3 and 4.3.3) assume a weakly synchronous network in which an attacker may delay messages and race target shards by replaying pre-recorded messages. The attacks against the second phase of the atomic commit protocol (Section 4.2.4 and 4.3.4) do not make any such assumptions on the underlying network.

**Attacks Implementation**. We implemented a demo of the replay attacks against S-BAC (attacks against Atomix can be similarly implemented) in Java. The demo shows, in the context of a simple payment application that supports account creation and coin transfer, how the replay attacks described in this paper can be used to create coins out of thin air. We show that the attacks do not rely on any strict timing assumptions—the same entity could control the accounts of both payer and payee, as well as the client, and generate coins out of thin air.

**Notation**. Operations on the blockchain are specified as *transactions*. A transaction defines some transformation on the blockchain state, and has input and output *objects* (such as UTXO entries). An object is some data managed by the blockchain, such as a bank account, a specific coin, or a hotel room. For example, $T(x_1, x_2) \rightarrow (y_1, y_2, y_3)$ represents a transaction with two inputs, $x_1$ managed by shard 1 and $x_2$ managed by shard 2; and three outputs, $y_1$ managed by shard 1, $y_2$ managed by shard 2, and $y_3$ managed by shard 3. We call the shards that manage the input objects *input shards*, and the shards that manage the output objects *output shards*. It is possible for a shard to be both the input and output shard. Objects can be in two states: *active* (on unspent) objects are available for being processed by a transaction; and *inactive* (or spent) objects cannot be processed by any transaction. Additionally, some systems also associate *locked* state with objects that are currently being processed by a transaction to protect against manipulation by other concurrent transactions involving those objects. The attacks we describe in this paper generalize to transactions with $k$ inputs and $k'$ outputs managed by an arbitrary number of shards.

## 4.2 Shard-led Cross-Shard Consensus Protocol

In shard-led cross-shard consensus protocols, the shards collectively take on the role of the coordinator in the atomic commit protocol. We describe replay attacks on shard-led cross-shard consensus protocols. To make the discussion concrete, we illustrate these attacks in the context of S-BAC [25], though we note that these attacks can be generalized to other similar systems. We discuss how the attacker can record shard messages to replay in future attacks (Section 4.2.2). In Sections 4.2.3 and 4.2.4, we describe replay attacks on the first and second phase of the cross-shard consensus protocol, and discuss the real-world impact of these attacks (Section 4.2.5).

### 4.2.1 S-BAC Overview

Chapter 3 presents Chainspace by treating the cross-shard consensus protocol as a black box. A preliminary version of Chainspace [25] relied on a shard-led cross-shard consensus protocol called S-BAC, which was insecure; since then, Chainspace has been updated to use the protocol described in Section 4.6.

In S-BAC, the client submits a transaction to the input shards. Each shard internally runs a BFT protocol to tentatively decide whether to accept or abort the transaction locally, and broadcasts its local decision (pre-accept($T$) or pre-abort($T$)) to other relevant shards. Figure 4.2 shows the state machine representing the life cycle of objects. A shard generates pre-abort($T$) if the transaction fails local checks (*e.g.*, if any of the input objects are 'inactive' or *locked*). If a shard generates pre-accept($T$), it changes the state of the input objects to *locked*. This is the first step of S-BAC, and is equivalent to the voting phase in the two-phase atomic commit protocol (Section 2.3).

Each shard collects responses from other relevant shards, and commits the transaction if all shards respond with pre-accept($T$), or aborts the transaction otherwise. This is the second step of S-BAC, and is equivalent to the commit phase in the two-phase atomic commit protocol (Section 2.3). The shards communicate this decision to the client as well as the output shards by sending them the accept($T$) or abort($T$) messages. If the shard's decision is accept($T$), it changes the input object state to 'inactive'. If the shard's decision is abort($T$), it changes the input object state

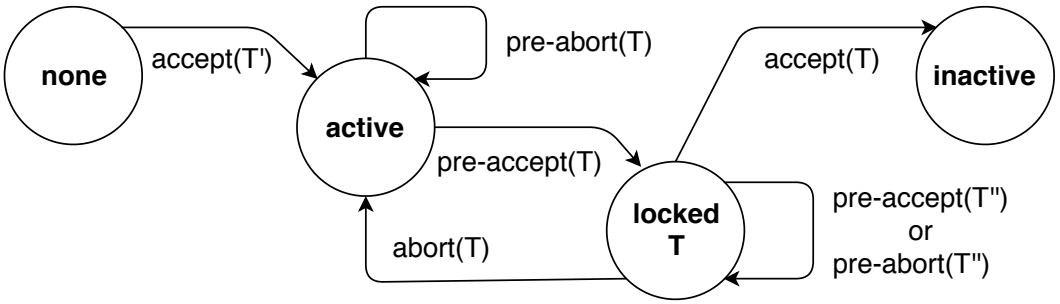


**Figure 4.2:** State machine representing the life cycle of objects handled by S-BAC. An object becomes 'active' as a result of a previous successful transaction. The object state changes to *locked* if a shard locally emits pre-accept($T$) in the first phase of the cross-shard consensus protocol for a transaction $T$. A locked object cannot be processed by other transactions $T''$. If the second phase of the protocol results in accept($T$), the object becomes 'inactive'; alternatively, if the result is abort($T$) the object becomes 'active' again and is available for being processed by other transactions.

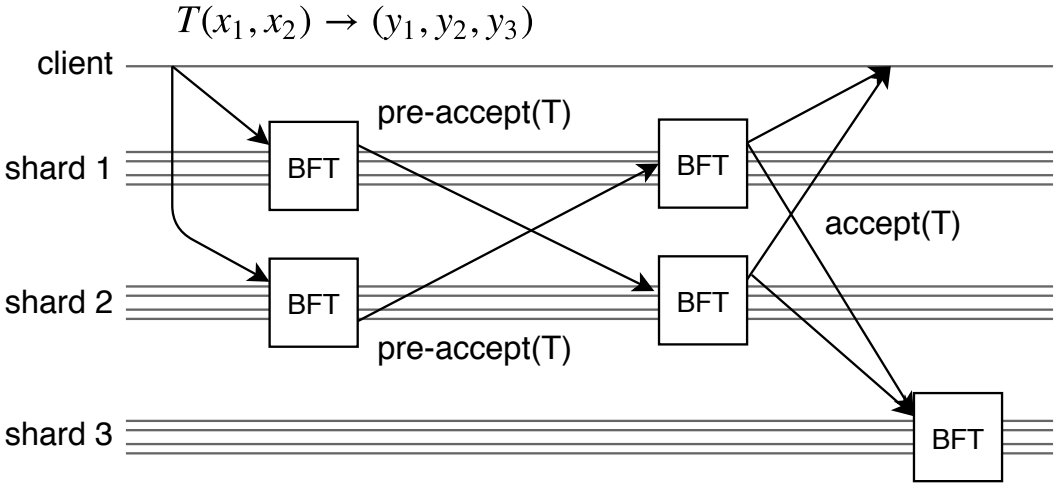


**Figure 4.3:** An example execution of S-BAC for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two inputs ($x_1$ and $x_2$, both are active) and three outputs ($y_1,y_2,y_3$), where the final decision is accept($T$). All cross-shard arrows represent a multicast of all nodes in one shard to all nodes in another.

to 'active' (effectively unlocking it). Upon receiving accept($T$), the client concludes that the transaction was committed, and the output shards create the output objects (with the state 'active') of the transaction.

Figure 4.3 shows an example execution of S-BAC for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two inputs ($x_1$ and $x_2$, both are active) and three outputs ($y_1,y_2,y_3$), where the final decision is accept($T$). The client submits $T$ to shard 1 and shard 2. Upon receiving $T$, both shard 1 and shard 2 confirm that the transaction is to commit, and emit pre-accept($T$) at the end of the first phase of S-BAC. Each

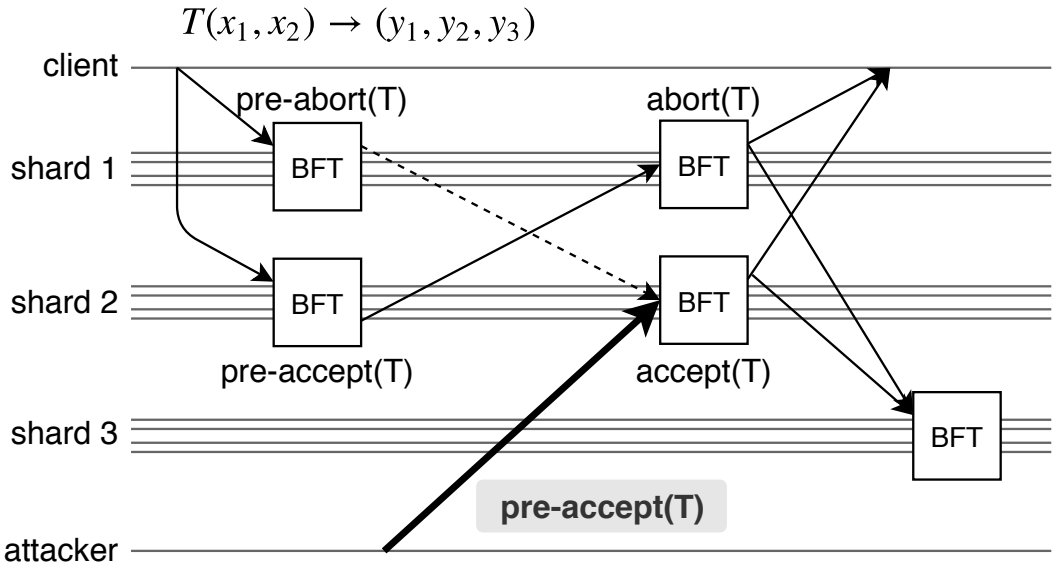


**Figure 4.4:** Illustration of the replay attack depicted in row 6 of Table 4.1. The attacker replays to shard 2 a prerecorded pre-accept($T$) message (shown as a bold line) from shard 1, which precludes shard 1's pre-abort($T$) message (shown as a dotted line).

shard receives pre-accept($T$) from the other shard, and emits accept($T$) at the end of the second phase of S-BAC. As a result, the input objects $x_1$ and $x_2$ become inactive, and the output shards respectively create objects $y_1$, $y_2$, and $y_3$.

### 4.2.2 Message Recording

Prior to the replay attacks, the attacker records responses generated by shards. The attacker can record shard responses in the first phase of S-BAC (*i.e.*, pre-accept($T$) or pre-abort($T$)), enabling the family of attacks described in Section 4.2.3. The attacker can also record shard responses in the second phase of S-BAC (*i.e.*, accept($T$) or abort($T$)), enabling the family of attacks described in Section 4.2.4. In the general case, the attacker passively collects the messages either by sniffing the network on protocol executions, or by downloading the blockchain and selecting the messages to replay[2]. Section 4.4.1 shows how the attacker can act as client to actively elicit the messages necessary for the attacks, to record and later replay—this empowers the attacker to actively orchestrate the attacks.

| | Phase 1 of S-BAC | | Phase 2 of S-BAC | | |
|---|---|---|---|---|---|
| | **Shard 1** (potential victim) | **Shard 2** (potential victim) | **Shard 1** (potential victim) | **Shard 2** (potential victim) | **Shard 3** (potential victim) |
| 1 | pre-accept($T$)<br>lock $x_1$ | pre-accept($T$)<br>lock $x_2$ | accept($T$)<br>create $y_1$; inactivate $x_1$ | accept($T$)<br>create $y_2$; inactivate $x_2$ | -<br>create $y_3$ |
| 2 | ▷pre-abort($T$) | | accept($T$)<br>create $y_1$; inactivate $x_1$ | abort($T$)<br>unlock $x_2$ | -<br>create $y_3$ |
| 3 | | ▷pre-abort($T$) | abort($T$)<br>unlock $x_1$ | accept($T$)<br>create $y_2$; inactivate $x_2$ | -<br>create $y_3$ |
| 4 | ▷pre-abort($T$) | ▷pre-abort($T$) | abort($T$)<br>unlock $x_1$ | abort($T$)<br>unlock $x_2$ | - |
| 5 | pre-abort($T$)<br>- | pre-accept($T$)<br>lock $x_2$ | abort($T$)<br>- | abort($T$)<br>unlock $x_2$ | - |
| 6 | ▷pre-accept($T$) | | abort($T$)<br>- | accept($T$)<br>create $y_2$; inactivate $x_2$ | -<br>create $y_3$ |
| 7 | pre-accept($T$)<br>lock $x_1$ | pre-abort($T$)<br>- | abort($T$)<br>unlock $x_1$ | abort($T$)<br>- | - |
| 8 | | ▷ pre-accept($T$) | accept($T$)<br>create $y_1$; inactivate $x_1$ | abort($T$)<br>- | -<br>create $y_3$ |
| 9 | pre-abort($T$)<br>- | pre-abort($T$)<br>- | abort($T$)<br>- | abort($T$)<br>- | - |

**Table 4.1:** List of replay attacks against the first phase of S-BAC for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 4.1. The highlighted rows indicate correct executions of S-BAC (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 3, row 2) means that shard 1 emits accept($T$) (top sub-row), and creates a new object $y_1$ and inactivates $x_1$ (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the first phase of S-BAC. The attacker races shards at the end of the first phase of S-BAC by replaying prerecorded messages, marked with the symbol ▷ in the first two columns of Table 4.1. For example ▷pre-abort($T$) at (column 1, row 2) means that the attacker sends to other relevant shards (in this case shard 2) a prerecorded pre-abort($T$) message impersonating shard 1 that races the original pre-accept($T$) (column 1, row 1) emitted by shard 1. The last three columns indicate the messages emitted at the end of the second phase of S-BAC.

### 4.2.3 Attacks on the First Phase of S-BAC

We present replay attacks on the first phase of S-BAC by taking the example of a transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 4.1. These attacks easily generalize to transactions with $k$ inputs and $k'$ outputs managed by an arbitrary number of shards. The replay attacks work in two steps; (i) the attacker records pre-accept($T$) or pre-abort($T$) messages (as described in Section 4.2.2 and Section 4.4.1); and (ii) then replays those messages. Table 4.1 shows the replay attacks that the

[2]Since those messages need to be recorded on chain for verification, just using transport layer encryption between nodes is not effective.

attacker can launch, for all possible combinations of messages emitted by shard 1 and shard 2 in the first phase of S-BAC. The caption includes details about how to interpret this table. All attacks exploit the parallel composition of multiple S-BAC instances, and insufficient binding of messages to its S-BAC instance. We describe row 6 of Table 4.1, to help readers interpret rest of the table on their own. In the correct execution (row 5), shard 1 and shard 2 emit pre-abort($T$) (because $x_1$ is not active) and pre-accept($T$) in the first phase, respectively. In the second phase, both shards emit abort($T$) and the protocol terminates. Figure 4.4 illustrates the replay attack corresponding to row 6 of Table 4.1. The attacker races shard 1 by sending to shard 2 the prerecorded pre-accept($T$) message from shard 1. As a result, shard 2 emits accept($T$), inactivates object $x_2$ and creates object $y_2$. This leads to inconsistent state across the shards. In a correct execution: (i) if $T$ is accepted all its inputs ($x_1$ and $x_2$) should become inactive, and all the outputs ($y_1$, $y_2$, $y_3$) should be created; and (ii) if $T$ is aborted, all its inputs ($x_1$ and $x_2$) should become active again, and none of the outputs ($y_1$, $y_2$, $y_3$) should be created. However, here we have an incorrect termination of S-BAC: at the end of the protocol $x_1$ could be active and $x_2$ is inactive; $y_1$ is not created, $y_2$ and $y_3$ are created.

Table 4.1 shows that through careful selection of the messages to replay from different S-BAC instances, the attacks can be effective against any shard. All the attacks (except row 4) compromise consistency; the attacker can trick the input shards to inactivate arbitrary objects, and trick the output shards into creating new objects in violation of the protocol. The attack depicted in row 4 only affects availability.

### 4.2.4 Attacks on the Second Phase of S-BAC

We present replay attacks on the second phase of S-BAC. The attacker prerecords accept($T$) messages as described in Section 4.2.2 and Section 4.4.1. Table 4.2 shows replay attacks for all possible combinations of messages emitted by shard 1 and shard 2 in the second phase. Since the attacks we describe in this section assume that the first phase of S-BAC concluded correctly (*i.e.*, all the relevant shards unanimously decide to accept or reject a transaction), both the shards generate abort($T$) (row 1) or accept($T$) (row 5). The caption includes details about how to interpret this table. We

| | Phase 2 of S-BAC | | |
|---|---|---|---|
| | **Shard 1** | **Shard 2** | **Shard 3** (potential victim) |
| 1 | accept($T$)<br>create $y_1$; inactivate $x_1$ | accept($T$)<br>create $y_2$; inactivate $x_2$ | -<br>create $y_3$ |
| 2 | ▷accept($T$) | | create $y_3$ |
| 3 | | ▷accept($T$) | create $y_3$ |
| 4 | ▷accept($T$) | ▷accept($T$) | create $y_3$ |
| 5 | abort($T$)<br>(unlock $x_1$) | abort($T$)<br>(unlock $x_2$) | -<br>- |
| 6 | ▷accept($T$) | | create $y_3$ |
| 7 | | ▷accept($T$) | create $y_3$ |
| 8 | ▷accept($T$) | ▷accept($T$) | create $y_3$ |

**Table 4.2:** List of replay attacks against the second phase of S-BAC for all possible executions of the transaction $T(x_1,x_2) \to (y_1,y_2,y_3)$ as described in Section 4.1. The highlighted rows indicate correct executions of S-BAC (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 1, row 1) means that shard 1 emits accept($T$) (top sub-row), and creates a new object $y_1$ and inactivates $x_1$ (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the second phase of S-BAC, and the last column shows the effect of these messages on the output shard 3. Replayed messages are marked with the symbol ▷. For example ▷accept($T$) at (column 1, row 2) means that the attacker sends to other relevant shards (in this case shard 3) a prerecorded accept($T$) message impersonating shard 1.

describe row 6 of Table 4.2, to help readers interpret rest of the table on their own. In the correct execution (row 5), both shards emit abort($T$) and no output objects are created. In the attack in row 6, the attacker replays a prerecorded accept($T$) from shard 1 to all the relevant shards (in this case shard 3). Upon receiving this message, shard 3 (incorrectly) creates $y_3$.

The potential victims of replay attacks corresponding to the second phase of S-BAC are the shards that *only* act as output shards (*i.e.*, do not simultaneously act as input shards). The attacker can replay accept($T$) multiple times tricking shard 3 into creating $y_3$ multiple times. These attacks are possible because shards do not keep records of inactive objects (following the UTXO model) for scalability reasons[3], and because shard 3 takes part in only the second phase of S-BAC. The attacker can double-spend $y_3$ repeatedly by replaying a single prerecorded message

[3]Requiring shards to remember the full history of inactive objects would increase their memory requirements monotonically over time, reaching at some point memory limits preventing further operations. Thus this is a poor mitigation for the attacks presented.

multiple times, and spending the object (and effectively purging it from shard 3's UTXO) before each replay. Contrarily to the attacks against the first phase of S-BAC (Section 4.2.3), these attacks do not rely on any racing conditions; there is no need to race any honest messages.

### 4.2.5 Real-world Impact

The real-world impact and attacker incentives to conduct these attacks depends on the nature and implementation of the smart contract handling the target objects. We discuss the impact of these attacks in the context of two common smart contract applications, which are also described in the Chainspace paper [25]. To take a concrete example, we illustrate the attack depicted in row 3 of Table 4.1, but similar results can be obtained with the other attacks described in Table 4.1 and Table 4.2.

One of the most common blockchain application is to manage cryptocurrency (or coins) and enable payments for processing transactions, implemented by the CSCoin smart contract in Chainspace. Lets suppose object $x_1$ (handled by shard 1) represents Alice's account, and object $x_2$ (handled by shard 2) represents Bob's account. To transfer $v$ coins to Bob, Alice submits a transaction $T(x_1,x_2) \rightarrow (y_1,y_2)$, where $y_1$ and $y_2$ respectively represent the new account objects of Alice and Bob, with updated account balances. By executing the attack described in row 3 of Table 4.1, an attacker can trick shard 1 to abort the transaction and unlock $x_1$ (thus reestablishing Alice's account balance as it was prior to the coin transfer), and shard 2 to accept the transaction and create $y_2$ (thus adding $v$ coins to Bob's account). This attack effectively allows any attacker to double-spend coins on the ledger; and shows how to create $v$ coins out of thin air. Another common blockchain use case is a platform for decision making (or electronic petitions), implemented by the SVote smart contract in Chainspace. Upon initialization, the SVote contract creates two objects: (i) $x_1$ representing the tally's public key, a list of all voters' public keys, and the tally's signature on these; and (ii) $x_2$ representing a vote object at the initial stage of the election (all candidates having a score of zero) along with a zero-knowledge proof asserting the correctness of the initial stage. To vote, clients submit a transaction $T(x_1,x_2) \rightarrow (y_1,y_2)$, where $y_1$ and $y_2$ are respectively the updated voting list (*i.e.*,

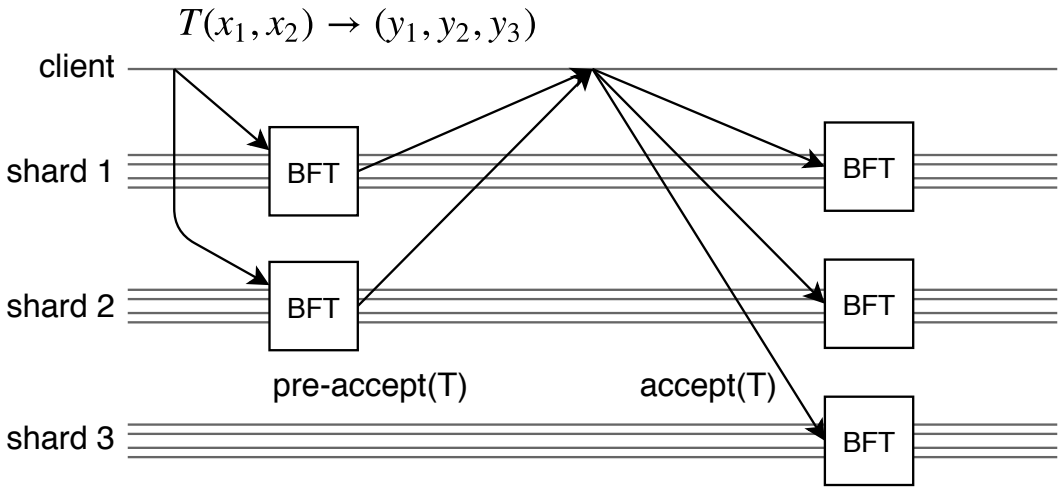


**Figure 4.5:** An example execution of Atomix for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ with two inputs ($x_1$ and $x_2$, both are active) and three outputs $(y_1,y_2,y_3)$, where the final decision is accept($T$).

the voting list without the client's public key), and the election stage updated with the client's vote. By executing the attack described by row 3 of Table 4.1, an attacker can trick shard 1 to abort the transaction and thus not update the voting list, and shard 2 to accept the transaction and thus update the election stage. This effectively allows any client to vote multiple times during an election while remaining undetected (due to the privacy-preserving properties of the SVote smart contract).

## 4.3 Client-led Cross-shard Consensus Protocol

We describe replay attacks on client-led cross-shard consensus protocols. To make the discussion concrete, we illustrate these attacks in the context of Atomix, the cross-shard protocol at the heart of Omniledger [4]. However, we note that these attacks can be generalized to other similar systems. We discuss how the attacker can record shard messages to replay in future attacks (Section 4.3.2). In Sections 4.3.3 and 4.3.4, we describe replay attacks on the first and second phase of the cross-shard consensus protocol. Finally, we discuss the real-world impact of these attacks (Section 4.3.5).

### 4.3.1 Atomix Overview

Similar to S-BAC, Atomix uses an atomic commit protocol to process transactions across shards. However, it uses a different, client-driven approach to achieve it. The

client submits the transaction $T$ to the input shards. Each shard runs a BFT protocol locally to decide whether to accept or reject the transaction, and communicates its response (pre-accept($T$) or pre-abort($T$)) to the client.[4] A shard emits pre-abort($T$) if the transaction fails local checks (*e.g.*, if any of the input objects is inactive). Alternatively, if a shard emits pre-accept($T$), it inactivates the input objects it manages. This is the first phase of Atomix, and is similar to the voting phase in the two-phase atomic commit protocol, but differs in that the protocol proceeds optimistically. The write changes made by the input shards in the first phase of Atomix are considered permanent (*i.e.*, there is no *locked* object state), unless the client requests the input shards to revert their changes in the second phase. After the client has collected pre-accept($T$) from all input shards, it submits accept($T$) message (containing proof of the pre-accept($T$) messages) to the output shards which create the output objects. Alternatively, if any of the input shards emits pre-abort($T$), the client sends abort($T$) (containing proof of pre-abort($T$)) to the relevant input shards which make the input objects active again. This is the second phase of Atomix, and is similar to the commit phase in the two-phase atomic commit protocol. Figure 4.5 shows the execution of Atomix for a valid transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$, with two active inputs ($x_1$ managed by shard 1, and $x_2$ managed by shard 2) and producing three outputs $(y_1,y_2,y_3)$ managed by shard 1, shard 2 and shard 3, respectively. The client sends $T$ to the input shards, both of which reply with pre-accept($T$) and make the input objects $x_1$ and $x_2$ inactive. The client sends accept($T$) to the output shards which respectively create objects $y_1$, $y_2$, and $y_3$.

### 4.3.2 Message Recording

Before launching the replay attacks, the attacker first records the target shard responses. The attacker can record shard responses in the first phase of Atomix (*i.e.*, pre-accept($T$) or pre-abort($T$)), enabling the attacks described in Section 4.3.3. The attacker can also record shard responses in the second phase of Atomix (*i.e.*, accept($T$) or abort($T$)), enabling the attacks described in Section 4.3.4. In the general

[4]For clarity and consistency, we use the terminology used in Chapter 3. In Atomix, pre-accept($T$) is actually a *proof-of-accept* and pre-abort($T$) is a *proof-of-abort* [4].

| | Phase 1 of Atomix | | | Phase 2 of Atomix | | |
|---|---|---|---|---|---|---|
| | **Shard 1** (potential victim) | **Shard 2** (potential victim) | **Client** (victim) | **Shard 1** (potential victim) | **Shard 2** (potential victim) | **Shard 3** (potential victim) |
| 1 | pre-accept($T$)<br>inactivate $x_1$ | pre-accept($T$)<br>inactivate $x_2$ | accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |
| 2 | ▷ pre-abort($T$) | | abort($T$) | -<br>re-activate $x_1$ | -<br>re-activate $x_2$ | - |
| 3 | | ▷ pre-abort($T$) | abort($T$) | -<br>re-activate $x_1$ | -<br>re-activate $x_2$ | - |
| 4 | ▷pre-abort($T$) | ▷pre-abort($T$) | abort($T$) | -<br>re-activate $x_1$ | -<br>re-activate $x_2$ | - |
| 5 | pre-abort($T$)<br>- | pre-accept($T$)<br>inactivate $x_2$ | abort($T$) | -<br>- | -<br>re-activate $x_2$ | - |
| 6 | ▷pre-accept($T$) | | accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |
| 7 | pre-accept($T$)<br>inactivate $x_1$ | pre-abort($T$)<br>- | abort($T$) | -<br>re-activate $x_1$ | -<br>- | - |
| 8 | | ▷ pre-accept($T$) | accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |
| 9 | pre-abort($T$)<br>- | pre-abort($T$)<br>- | abort($T$) | -<br>- | -<br>- | - |
| 10 | ▷ pre-accept($T$) | ▷ pre-accept($T$) | accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |

**Table 4.3:** List of replay attacks against the first phase of Atomix for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 4.1. The highlighted rows indicate correct executions of Atomix (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions as a result of emitting those messages. For example, (column 1, row 1) means that shard 1 emits pre-accept($T$) (top sub-row), and inactivates $x_1$ (bottom sub-row). The first two columns indicate the messages emitted by each shard at the end the first phase of Atomix. Replayed messages are marked with the symbol ▷, for example ▷pre-abort($T$) at (column 1, row 2) means that the attacker sends to the client a prerecorded pre-abort($T$) message impersonating shard 1 that races the original pre-accept($T$) (column 1, row 1) emitted by shard 1. The third column indicates the messages sent by the client to the relevant shards, and the last three columns indicate the local actions performed by shards at the end of the second phase of Atomix.

case, the attacker passively collects the messages to replay, for example by protocol executions on the network, or by downloading the blockchain and selecting the appropriate messages. Section 4.4.2 shows how the attacker can act as a client to actively elicit and record the target messages to later use in the replay attacks.

### 4.3.3 Attacks on the First Phase of Atomix

We present replay attacks on the first phase of Atomix by taking the example of a transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 4.1. These attacks easily

generalize to transactions with $k$ inputs and $k'$ outputs managed by an arbitrary number of shards. The replay attacks work in two steps: (i) the attacker observes the traffic and records pre-accept($T$) or pre-abort($T$) messages as described in Section 4.3.2; and (ii) then replay those messages. Table 4.3 shows the replay attacks that the attacker can launch, for all possible combinations of responses generated by shard 1 and shard 2 in the first phase of Atomix. The caption includes details about how to interpret this table. We describe row 6 of Table 4.3, to help readers interpret rest of the table on their own. In the correct execution (row 5), shard 1 emits pre-abort($T$), and shard 2 emits pre-accept($T$) and inactivates the input objects $x_2$. Upon receiving these messages, the client sends abort($T$) to the output shards shard 1, shard 2 and shard 3, and shard 2 re-activates $x_2$; and the protocol terminates. In the attack illustrated in row 6 of Table 4.3, the attacker races shard 1 by sending to the client the prerecorded pre-accept($T$) message from shard 1. As a result, the client sends accept($T$) message to the output shards shard 1, shard 2 and shard 3, which respectively create the output objects $y_1$, $y_2$, and $y_3$. As a result, the system ends up in an inconsistent state because the output objects ($y_1$, $y_2$, $y_3$) have been created, while the input object ($x_1$) was not active—this results in a double-spend of the input object $x_1$. Table 4.3 shows that through careful selection of the messages to replay, the attacks can be effective against any shard. The attacks illustrated in row 2, row 3, and row 4 only affect availability, while the other attacks compromise consistency (*i.e.*, the attacker can trick the input shards to reactivate arbitrary objects, and trick the output shards into creating new objects in violation of the protocol). The potential victims of these attacks include the client (*e.g.*, when the attacker replays the shard messages to it in the first phase of Atomix) and any input or output shards.

### 4.3.4 Attacks on the Second Phase of Atomix

We present replay attacks on the second phase of Atomix. The attacker prerecords accept($T$) and abort($T$) messages as described in Section 4.3.2 and Section 4.4.2. Table 4.4 shows replay attacks corresponding to the messages emitted by the client in the second phase—*i.e.*, accept($T$) in row 1, or abort($T$) in row 3. The caption includes details about how to interpret this table. The abort($T$) message at (column

| | Client | Phase 2 of Atomix<br>**Shard 1**<br>(potential victim) | **Shard 2**<br>(potential victim) | **Shard 3**<br>(potential victim) |
|---|---|---|---|---|
| 1 | accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |
| 2 | ▷abort($T$) | -<br>re-activate $x_1$ | -<br>re-activate $x_2$ | - |
| 3 | abort($T$) | -<br>re-activate $x_1$ | -<br>re-activate $x_2$ | - |
| 4 | ▷accept($T$) | -<br>create $y_1$ | -<br>create $y_2$ | -<br>create $y_3$ |

**Table 4.4:** List of replay attacks against the second phase of Atomix for all possible executions of the transaction $T(x_1,x_2) \rightarrow (y_1,y_2,y_3)$ as described in Section 4.1. The highlighted rows indicate correct executions of Atomix (*i.e.*, without the attacker), and the other rows indicate incorrect executions due to the replay attacks. In multirows, the top sub-rows show the protocol messages emitted by shards, and the bottom sub-rows indicate local shard actions. Note that we use the multirow format for consistency reasons; in this table the first column indicates the messages emitted by the client at the beginning of the second phase of Atomix, and the last two column shows the effect of these messages on the relevant shards. Replayed messages are marked with the symbol ▷. For example, ▷abort($T$) at (column 1, row 2) means that the attacker sends a prerecorded abort($T$) message to the input shards impersonating the client.

1, row 2) means that the attacker sends a prerecorded abort($T$) message to the input shards (shard 1 and shard 2) impersonating the client. Upon receiving this message, shard 1 and shard 2 (incorrectly) re-activate $x_1$ and $x_2$, respectively. Furthermore, all output shards create the output objects when the correct accept($T$) message emitted by the client (row 1, column 1) reaches them. This results in inconsistent state, because the output objects have been created, but the input objects have not been consumed and have been reactivated by the abort($T$) message replayed by the adversary. The potential victims of abort($T$) replay attack are the input shards. Similarly, accept($T$) at (row 4, column 1) means that the attacker sends a prerecorded accept($T$) message to the output shards (shard 1, shard 2 and shard 3) impersonating the client. Upon receiving this message, the output shards (incorrectly) create $y_1$, $y_2$ and $y_3$. Furthermore, the input shards (shard 1 and shard 2) reactivate $x_1$ and $x_2$ upon receiving the correct abort($T$) message emitted by the client (row 3, column 1). This creates inconsistent state: the input objects have not been consumed and have been reactivated by the abort($T$) message emitted by the client, but the output objects have been created due to the accept($T$) message replayed by the attacker.

The potential victims of accept($T$) replay attack are the output shards.

These attacks are possible because output shards create objects directly upon receiving accept($T$); they do not check if the objects have been previously invalidated because shards do not keep records of inactive objects (per the UTXO model) for scalability reasons.[5] The attacker can double-spend the output objects repeatedly from a single prerecorded message by replaying it multiple times, and spending the object (and effectively purging it from the output shards' UTXO) before each replay. Similar to the attacks against the second phase of S-BAC (Section 4.2.4), these attacks do not exploit any racing condition and can be mounted by an adversary at a leisurely pace.

### 4.3.5 Real-world Impact

Contrarily to Chainspace, Omniledger does not support smart contracts and only handles a cryptocurrency. The attacks described in Sections 4.3.3 and 4.3.4 allow an attacker to: (i) double-spend the coins of any user, by reactivating spent coins (*e.g.*, the attacker may execute the attack depicted by row 2 of Table 4.4 to re-activate the objects $x_1$ and $x_2$ after the transfer is complete); and (ii) create coins out of thin air by replaying the message to create coins (*e.g.*, an attacker may execute the attack depicted by row 4 of Table 4.4 to create multiple times object $y_3$, by purging it from the UTXO list of shard 3 prior to each instance of the attack). If the attacker colludes with the client, it can trigger the prerecorded messages needed for the attacks as described in Section 4.3.2. Alternatively, the attacker can passively observe the network and collect the target messages to replay. Similar results can be obtained using the attacks described in Table 4.3. Note that since transaction are recorded on the blockchain, these attacks can be detected retrospectively. This can lead to the attacker being exposed, or the attacker can inculpate innocent users (the attacker can replay messages of any user).

[5] Verifying that objects have not been previously invalided implies either keep a forever-growing list of invalidated objects, or download and check the shard's entire blockchain.

## 4.4 Eliciting Messages to Replay

We show how the attacker can act as (or collude with) a client to actively elicit and record the target messages to later use in the replay attacks. This empowers the attacker to actively orchestrate the attacks. We describe how the attacker can trigger target messages in the context of an example, without loss of generality. Lets assume that shard 1 manages objects $x_1$ ('active') and object $\widetilde{x_1}$ ('inactive' or non-existent), and shard 2 manages object $x_2$ ('active'); $\widetilde{x*}$ means any inactive object on the shard, and $y*$ means any output object (*i.e.*, their details do not matter).

### 4.4.1 Shard-led Cross-Shard Consensus

We show how the attacker can act as (or collude with) a client to actively elicit and record the target messages, in the context of shard-led cross-shard consensus protocols as illustrated by Section 4.2. To elicit pre-accept($T$) for a transaction $T(x_1,x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from shard 1, the key consideration is to closely precede the transaction with another transaction $T'$ that: (i) locks the inputs managed by at least one other shard (in this case $x_2$ on shard 2); and (ii) to ensure that the preceding transaction $T'$ gets ultimately aborted, and $x_2$ becomes active again. The steps look as follows:

- The attacker submits $T'(x_2,\widetilde{x*}) \rightarrow (y*)$ to shard 2. This locks $x_2$.
- The attacker quickly follows up by submitting $T(x_1,x_2) \rightarrow (y*)$ to shard 1 and shard 2. Shard 1 generates pre-accept($T$), which is the target message that the attacker records. Shard 2 generates pre-abort($T$) because $x_2$ is locked by $T'$. Consequently, in the second phase of S-BAC, both shard 1 and shard 2 end up aborting $T$.
- $T'$ is eventually aborted, making $x_2$ active again.

To elicit pre-abort($T$) for a transaction $T(x_1,x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from shard 1, the key consideration is to closely precede the transaction with another transaction $T'$ that locks the input managed by the shard (in this case $x_1$ on shard 1). The steps look as follows:

- The attacker submits $T'(x_1, \widetilde{x*}) \rightarrow (y*)$ to shard 1. This locks $x_1$.

- The attacker quickly follows up by submitting $T(x_1, x_2) \rightarrow (y*)$ to shard 1 and shard 2. Shard 1 generates pre-abort($T$) because $x_1$ is locked by $T'$, which is the target message that the attacker records. Shard 2 generates pre-accept($T$). Consequently, in the second phase of S-BAC, both shard 1 and shard 2 end up aborting $T$.

- $T'$ is eventually aborted, making $x_1$ active again.

To elicit accept($T$) used by the attacks described in Section 4.2.4, the attacker simply submits transaction $T$ and observes and records its successful execution. The attacker has no incentive to record abort($T$) messages as these are ignored by shards (see Table 4.2).

### 4.4.2 Client-led Cross-Shard Consensus

We show how the attacker can act as (or collude with) a client to actively elicit and record the target messages, in the context of client-led cross-shard consensus protocols as illustrated by Section 4.3. To elicit pre-accept($T$) from shard 1 for a transaction $T(x_1, x_2) \rightarrow (y*)$ (the output $y*$ is not relevant here) from shard 1, the key consideration is to closely precede the transaction with another transaction that: (i) temporarily spends the inputs managed by at least one other shard (in this case $x_2$ on shard 2); and (ii) to ensure that the preceding transaction is ultimately aborted so that $x_2$ becomes active again. The steps look as follows:

- The attacker submits $T'(x_2, \widetilde{x*}) \rightarrow (y*)$ to shard 2, where $\widetilde{x*}$ is managed by a different shard. shard 2 emits pre-accept($T'$) and marks $x_2$ as inactive.

- The attacker follows up by submitting $T(x_1, x_2) \rightarrow (y*)$ to shard 1 and shard 2. Shard 1 generates pre-accept($T$), which is the target message that the attacker records. Shard 2 generates pre-abort($T$) because $x_2$ is inactive.

- The attacker submits abort($T$) to shard 1 to reactivate $x_1$, and sends abort($T'$) to shard 2 to reactivate $x_2$.

For the attacks described in Section 4.3.4, the attacker needs to elicit abort($T$) and accept($T$) from the target shards. For the former, the attacker can follow the steps described previously to elicit pre-accept($T$) and pre-abort($T$). To elicit accept($T$), the attacker simply submits transaction $T$ and observes and records its successful execution.

## 4.5 Defenses Against Replay Attacks

We identify two issues that lead to the replay attacks described in Section 4.2 and Section 4.3, and discuss how to fix those:

- First, the input shards do not have a way to know that particular protocol messages received correspond to a specific instance (or session) of the protocol. This gap in the input shards' knowledge enables an attacker to replay, mix and match, old messages leading to attacks. To address this limitation, we associate a session identifier with each transaction, which has to be crafted carefully to not degrade the performance of the protocols significantly—such as by requiring nodes to store state linearly in the number of past transactions.

- Second, in some cases the output shards are only involved in the second phase of the protocol, and therefore have no knowledge of the transaction context (to determine freshness) that is available to the input shards. This limitation can be addressed by ensuring that all shards—input and output—witnesses the entire protocol execution, rather than just one of the protocol phases.

Note that the two mitigation techniques described above must be used together, as part of a single defense strategy against replay attacks.

## 4.6 The Byzcuit Atomic Commit Protocol

We showed that both S-BAC (Sections 4.2.3 and 4.2.4) and Atomix (Sections 4.3.3 and 4.3.4) are vulnerable to replay attacks that can compromise system liveness and safety. Atomix is the simpler protocol of the two, and using the client to coordinate cross-shard communication can reduce the cost to $O(n)$ in the number

of shards (by aggregating shard messages). However, an unresponsive or malicious client can permanently lock input objects by never initiating the second phase of the protocol, requiring additional design considerations (*e.g.*, a new entity that periodically unlocks input objects for transactions on which no progress has been made). On the other hand, S-BAC runs the protocol among the shards, without relying on client coordination. But this comes at the cost of increased cross-shard communication: all input shards communicate with all other input shards, which leads to communication complexity of $O(n^2)$ where $n$ is the number of input shards.

Motivated by these insights, we present Byzcuit—a cross-shard atomic commit protocol (based on S-BAC) that integrates design features from Atomix—and offers better performance and security against replay attacks. Byzcuit allocates a Transaction Manager (TM) to coordinate cross-shard communication, reducing its cost to $O(n)$ in the happy case[6]; alternatively Byzcuit also has a fall-back mode in case the TM fails, similar to Atomix and traditional two phase commit protocols. Byzcuit achieves resilience against the replay attacks described in Section 4.2 and Section 4.3, by leveraging the defense proposed in Section 4.5.

### 4.6.1 Byzcuit Protocol Design

We describe how Byzcuit integrates the defense presented in Section 4.3. To map particular protocol messages to a specific protocol instance (or session), Byzcuit associates a session identifier with each transaction. To ensure that all the relevant (input and output) shards witness all phases of the protocol execution, Byzcuit leverages the notion of *dummy objects*. Each shard creates a fixed number of dummy objects upon configuration; if a shard only serves as an output shard for a transaction, Byzcuit forces it to be involved in the first phase of the protocol by implicitly including a dummy object managed by the output shard in the transaction inputs, which will create a new dummy object upon completion. Thus, the output shard also becomes an input shard (because of the dummy object in the transaction inputs) and witnesses the entire protocol, rather than just the second phase.

[6]The communication complexity can be reduced to $O(n)$ in the number of shards by aggregating shard messages as described by Omniledger.

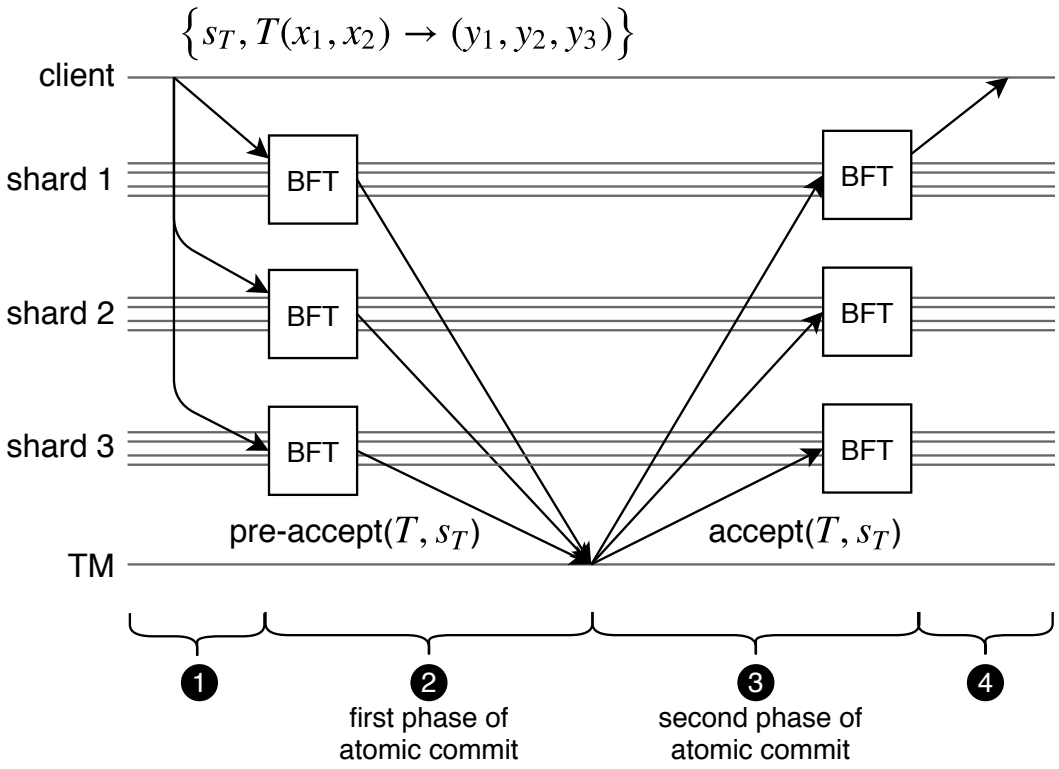


**Figure 4.6:** An example execution of Byzcuit for a valid transaction $T(x_1,x_2) \to (y_1,y_2,y_3)$ with two input objects ($x_1$ and $x_2$, both are active), and three outputs $(y_1,y_2,y_3)$, where the final decision is accept$(T,s_T)$.

**Byzcuit protocol execution**. We illustrate Byzcuit taking the example of a transaction $T(x_1,x_2) \to (y_1,y_2,y_3)$ with two input objects, $x_1$ managed by shard 1 and $x_2$ managed by shard 2; and three outputs, $y_1$ managed by shard 1, $y_2$ managed by shard 2, and $y_3$ managed by shard 3. Figure 4.6 illustrates the Byzcuit protocol; the client first sends the transaction to all input and output shards. Note that this is different than other protocols like S-BAC and Atomix, where the transaction is only sent to the input shards. As mentioned previously, to achieve resilience against replay attacks, Byzcuit forces a shard that is *only* involved in creating the output objects to also become an input shard (and witness the transnational context by participating in the first phase of the protocol) by implicitly consuming one of its dummy inputs (which creates a new dummy object upon completion). Byzcuit associates a sequence number $s_{x_i}$ to each object and dummy object (when the object is created $s_{x_i} = 0$). The sequence number is intrinsically linked to the object: when clients query shards to obtain an object $x_i$, they also receive the associated sequence number $s_{x_i}$.

When submitting the transaction $T$, the client also sends along a transaction sequence number $s_T = max\{s_{x_1}, s_{x_2}, s_{d_3}\}$, where the transaction sequence number $s_T$ is the maximum of the sequence numbers $s_{x_i}$ of each input object $x_i$ and dummy objects $d_i$ (❶). Upon receiving a new pair $(T, s_T)$, each shard saves $(T, s_T)$ in a local cache memory—the transaction sequence number $s_T$ acts as session identifier associated with the transaction $T$. Each shard internally verifies that the transaction passes local checks, and that $s_T$ is equal to (or bigger than) the sequence numbers of the objects they manage (*i.e.*, shard 1 checks $s_T \geq s_{x_1}$, shard 2 checks $s_T \geq s_{x_2}$, shard 3 checks $s_T \geq s_{d_3}$). The shards send their local decision to the TM: pre-accept$(T, s_T)$ for local accept (and the shard locks the objects it manages), or pre-abort$(T, s_T)$ for local abort. After receiving all the messages corresponding to the first phase of Byzcuit from the concerned shards, the TM sends a suitable message to the shards (accept$(T, s_T)$ if all the shards respond with pre-accept$(T, s_T)$, or abort$(T, s_T)$ otherwise). Upon receiving accept$(T, s_T)$ or abort$(T, s_T)$ from the TM, shards first verify that they previously cached the pair $(T, s_T)$ associated with the message; otherwise they ignore it (❷).

The accept$(T, s_T)$ or abort$(T, s_T)$ messages sent by the TM provide enough evidence to the shards to verify whether $s_T$ is correctly computed; *i.e.* shards verify that $s_T$ is at least the maximum of the sequence numbers of each input and dummy object by inspecting the transaction $T$ signed by each shard. If accept$(T, s_T)$ has a correct $s_T$, the shards inactivate the input objects and create the output objects $(y_1, y_2, y_3)$, and shard 3 creates a new dummy object $\widetilde{d_3}$; otherwise, they update the sequence numbers of each input object $(s_{x_1}, s_{x_2})$ and dummy object $d_3$ to $(s_T + 1)$, *i.e.* shards locally update $s_{x_1} \leftarrow (s_T + 1)$ and $s_{x_2} \leftarrow (s_T + 1)$, and $s_{d_3} \leftarrow (s_T + 1)$. Shards delete $(T, s_T)$ from their local cache (❸). Since we assume that shards are honest—inline with the threat model of the systems discussed—it suffices if only one shard notifies the client of the protocol outcome; we may set any arbitrary rule to decide which shard notifies the client (*e.g.*, the shard handling the first input object) (❹). Figure 4.7 shows the finite state machine describing the life cycle of Byzcuit objects.

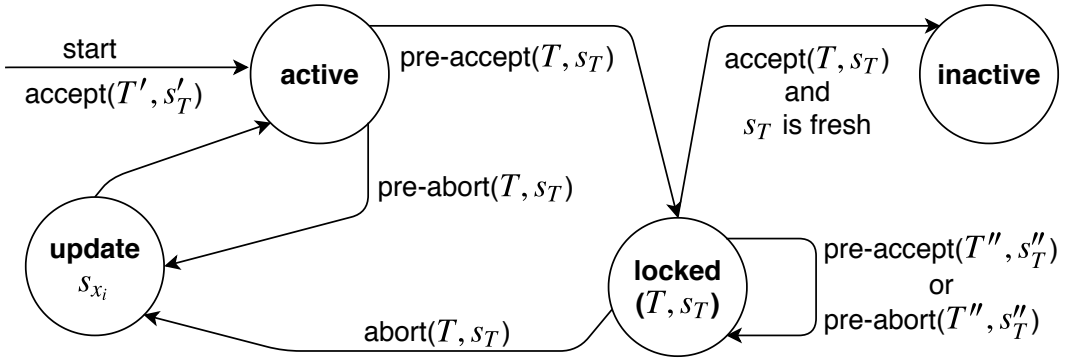


**Figure 4.7:** State machine representing the life cycle of objects in Byzcuit. Objects are initially 'active'. Upon receiving a transaction that passes local checks, a shard changes its input objects' state to *locked* (objects are locked for a given transaction $T$ and transaction sequence number $s_T$) and emits pre-accept$(T, s_T)$; otherwise it updates the sequence number of every object it manages and emits abort$(T, s_T)$. Once a shard locks input objects for a given $(T, s_T)$, any accept$(T, s_T)$ and abort$(T, s_T)$ with malformed transaction sequence numbers are ignored, and do not cause any transition (not included in the figure). Any incoming transaction $T'$ that requires processing *locked* input object(s) is aborted. Upon receiving accept$(T, s_T)$ with a well formed $s_T$, a shard makes its input objects 'inactive' and creates the output objects. Alternatively, upon receiving abort$(T, s_T)$ shards unlock their inputs and updates the corresponding sequence numbers.

**Transaction manager**. The Transaction Manager (TM) coordinates cross-shard communication in Byzcuit. We now discuss who might play the role of the TM, and argue that Byzcuit guarantees liveness even if the TM is faulty (Byzantine) or crashes. Keeping with the overall design goal of decentralization, we envision that a designated shard will act as the TM. If the shard is honest, the TM is live—and therefore progress is always made. The input shards contact in turn each node of the TM shard until they reach one honest node. The TM shard may have up to $f$ dishonest nodes; therefore, the client or the input shards need to send messages to at least $f+1$ nodes of the TM shard to ensure that it is received by at least one honest node[7]. Thus, as soon as the first honest node receives the message, the protocol is guaranteed to make progress. If the TM is the client or any centralized party, it may act arbitrarily—but this does not stall the protocol because anyone can make the protocol progress by taking over at any time the role of the TM. This is possible because the TM does not act on the basis of any secrets, therefore anyone else can

[7]Clients may take a statistical view of availability. Given that fewer than $2/3$ of nodes in a shard are dishonest, sending the transaction to $\rho$ nodes will fail to reach an honest node with probability only $(1/3)^\rho$. Clients may also send messages sequentially to nodes, and only continue if they do not observe progress within some timeout to further reduce costs.

take over and complete the protocols. This 'anyone' may be an honest node in a shard that wants to finally unlock an object (*e.g.*, upon a timeout); or other clients that wish to use a locked object; or it may be an external service that has a job to periodically close open Byzcuit instances. Byzcuit ensures such parties may attempt to make progress asynchronously and concurrently safely. Therefore, Byzcuit guarantees liveness as long as there is at least one honest entity in the system.

**Handling sequence number overflow**. An attacker can try to exhaust the possible sequence numbers to make them overflow. The attacker submits a pair $(T, s_T)$ such that the sequence number $s_T$ is just below the system overflow value; the sequence numbers associate with the inputs overflow upon the next updates, and the system would be again prone to the attacks described in Section 4.2.3[8]. To mitigate this issue, shards define a *clone* procedure allowing to update any of their objects to an unchanged version of themselves (*i.e.* it creates a fresh copy of the object). This clone procedure effectively creates a new object with serial number $s'_x = 0$. When shards detect that the serial number of one of their objects approaches the overflow value, they execute internally this clone procedure. The attacker may exploit this mechanism to DoS the system, forcing shards to constantly update their objects; as a result, the target objects are not available to users. DoS countermeasures are out of scope, and are typically addressed by introducing transaction fees.

### 4.6.2 Security against Replay Attacks

We argue that Byzcuit is resilient to replay attacks. We recall the Honest Shard assumption from Chainspace (Section 3.1.2) under which Byzcuit operates, and assume that messages are authenticated as in traditional BFT protocols.

**Security Assumption 1.** *(Honest Shard) The adversary may create arbitrary smart contracts, and input arbitrary transactions into Byzcuit, however they are bound to only control up to $f$ faulty nodes in any shard. As a result, and to ensure the correctness and liveness properties of Byzantine consensus, each shard must have a size of at least $3f+1$ nodes.*

[8]Note that this overflow vulnerability is common to every system relying on nonces chosen by the users, like Byzantine Quorum Systems [142].

Any message emitted by shards comes with at least $f+1$ signatures from nodes. Assuming honest shards, the attacker can forge at most $f$ signatures, which is not enough to impersonate a shard. We use the Lemma below to prove the security of Byzcuit through this section.

**Lemma 1.** *Under Honest Shard assumption, no attacker can obtain prerecorded messages containing a fresh transaction sequence number $s_T$.*

*Proof.* The key idea protecting Byzcuit from these replay attacks is that the attacker can only obtain prerecorded messages associated with old transaction sequence numbers $s_T$. The transaction sequence number $s_T$ is fresh only if it is at least equal the maximum of the sequence number of all input and dummy objects of the transaction $T$. Shards update every input and dummy object sequence number upon aborting transactions in such a way that sequence numbers only increase. That is, after emitting pre-abort$(T, s_T)$ or pre-accept$(T, s_T)$, either the sequence number of all input and dummy objects of $T$ are updated to a value bigger than $s_T$ (in case of pre-abort$(T, s_T)$), or the objects are inactivated which prevents any successive transaction to use them as input (in case of pre-accept$(T, s_T)$). It is thus impossible for the adversary to hold a prerecorded message for a fresh $s_T$; the only prerecorded messages that the adversary can obtain contain sequence numbers smaller than $s_T$. □

**Security of the first phase of Byzcuit**. An attacker may try to replay pre-accept$(T, s_T)$ and pre-abort$(T, s_T)$ during the first phase of the protocol, similarly to the attacks described in Sections 4.2.3 and 4.3.3; the TM then aggregates these messages into either accept$(T, s_T)$ or abort$(T, s_T)$, and forwards them to the shards during the second phase of the protocol. Security Theorem 4 shows that Byzcuit detects that they originate from replayed messages and ignores them. Intuitively, the transaction sequence number $s_T$ acts as a monotonically increasing session identifier associated with the transaction $T$; the attacker cannot obtain prerecorded messages containing a fresh $s_T$. Byzcuit shards can then distinguish replayed messages (*i.e.*, messages with old $s_T$) from the messages coming from the instance of the protocol that they are executing (*i.e.*, messages with fresh $s_T$).

**Security Theorem 4.** *Under Honest Shard assumption, Byzcuit ignores accept$(T, s_T)$ and abort$(T, s_T)$ messages issued from replayed pre-accept$(T, s_T)$*

*and pre-abort($T, s_T$).*

*Proof.* Figure 4.7 shows that once Byzcuit locks objects for a particular pair $(T, s_T)$, the protocol can only progress toward accept$(T, s_T)$ or abort$(T, s_T)$; *i.e.* shards can either accept or abort the transaction $T$. The attacker aims to trick one or more shards to incorrectly accept or abort $T$ by injecting prerecorded messages during the first phase of Byzcuit; we show that the attacker fails in every possible scenario.

Suppose transaction $T$ should abort (the TM outputs abort$(T, s_T)$), but the attacker tries to trick some shards to accept the transaction. Figure 4.7 shows that the attacker can only succeed the attack if they gather accept$(T, s_T)$ containing a fresh transaction sequence number $s_T$. Lemma 1 states that no attacker can obtain prerecorded messages over a fresh transaction sequence number $s_T$; therefore the only messages available to the adversary at this point of the protocol are (at most) $n-1$ pre-accept$(T, s_T)$ and (at most) $n$ abort$(T, s_T)$, where $n$ is the number of concerned shards. This is not enough to form an accept$(T, s_T)$ message with a fresh transaction sequence number $s_T$ (which is composed of $n$ pre-accept$(T, s_T)$); therefore the attacker cannot trick any shard to accept the transaction.

Suppose transaction $T$ should be accepted (the TM outputs accept$(T, s_T)$ with a fresh $S_T$), but the attacker tries to trick some shards to abort the transaction. Figure 4.7 show that Byzcuit does not require a fresh transaction sequence number $s_T$ to abort transactions (the freshness of $s_T$ is only enforced upon accepting a transaction); but shards locked the input and dummy objects of the transaction for the pair $(T, s_T)$ (with fresh $s_T$), so the attacker needs to gather abort$(T, s_T)$ containing the same transaction sequence number $s_T$ locked by shards. Lemma 1 shows that the attacker cannot obtain prerecorded messages over fresh $s_T$; therefore the only messages available to the adversary containing the (fresh) $s_T$ locked by shards at this point of the protocol are $n$ pre-accept$(T, s_T)$. This is not enough to form an abort$(T, s_T)$ message (which is composed of at least one pre-abort$(T, s_T)$); therefore the attacker cannot trick any shard to abort the transaction. □

**Security of the second phase of Byzcuit**. An attacker may try to replay accept$(T, s_T)$ and abort$(T, s_T)$ messages during the second phase of the protocol, similarly to the attacks described in Sections 4.2.4 and 4.3.4. Security Theorem 5 shows that Byzcuit ignores those replayed messages. Intuitively, these attacks target shards acting only as output shards (and not also as input shards) and exploit the fact

that they are only involved in the second phase of the protocol, and therefore have no knowledge of the transaction context (to determine freshness) that is available to the input shards. Byzcuit is resilient to these replay attacks as it is designed in such a way that there are no shards that act only as output shards; all output shards are forced to also become input shards, by introducing dummy objects if they do not manage any input objects; this prevents the attacks by removing the attack target.

**Security Theorem 5.** *Under Honest Shard assumption, Byzcuit ignores replayed accept($T, s_T$) and abort($T, s_T$) messages.*

*Proof.* Figure 4.7 shows that shards only act upon accept($T, s_T$) and abort($T, s_T$) messages if they have the pair $(T, s_T)$ saved in their local cache[9]. Shards save a pair $(T, s_T)$ in their local cache upon emitting pre-accept($T, s_T$) or pre-abort($T, s_T$), and delete it at the end of the protocol; therefore the only attack windows where the adversary can replay accept($T, s_T$) and abort($T, s_T$) messages is while the transaction $T$ (associated with $s_T$) is being processed by the second phase of Byzcuit. This forces the attacker to operates under the same conditions as Security Thereom 4, which is shown secure above. □

### 4.6.3 Byzcuit Security & Correctness

We show that Byzcuit guarantees liveness, consistency, and validity.

**Security Theorem 6.** *(Liveness) Under Honest Shards assumption, a transaction $T$ that is proposed to at least one honest concerned node, eventually results in either being committed or aborted, namely all parties deciding accept($T, s_T$) or abort($T, s_T$).*

*Proof.* We rely on the liveness properties of the Byzantine agreement (shards with only $f$ nodes eventually reach consensus on a sequence), and the broadcast from nodes of shards to all other nodes of shards, channelled through the Transaction Manager. Assuming $T$ has been given to an honest node, it will be sequenced withing an honest shard BFT sequence, and thus a pre-accept($T, s_T$) or pre-abort($T, s_T$) will be sent from the $2f + 1$ honest nodes of this shard, aggregated into accept($T, s_T$) or abort($T, s_T$), and sent to the $f + 1$ nodes of the other concerned shards. Upon receiving these messages the honest nodes from other

[9]Contrarily to S-BAC and Atomix, all Byzcuit shards have the pair $(T, s_T)$ in their local cache after as they all participate to the first phase of the protocol.

shards will process the transaction within their shards, and the BFT will eventually sequence it. Thus the user will eventually receive a decision from at least $f+1$ nodes of a shard. □

**Security Theorem 7.** *(Consistency) Under Honest Shards assumption, no two conflicting transactions, namely transactions sharing the same input will be committed. Furthermore, a sequential executions for all transactions exists.*

*Proof.* A transaction is accepted only if some nodes receive accept$(T, s_T)$, which presupposes all shards have provided enough evidence to conclude pre-accept$(T, s_T)$ for each of them. Two conflicting transaction, sharing an input, must share a shard of at least $3f+1$ concerned nodes for the common object—with at most $f$ of them being malicious. Without loss of generality upon receiving the pre-accept$(T, s_T)$ message for the first transaction, this shard will sequence it, and the honest nodes will emit messages for all—and will lock this object until the two phase protocol concludes. Any subsequent attempt to pre-accept$(T, s_T)$ for a conflicting $T'$ will result in a pre-abort$(T, s_T)$ and cannot yield a accept, if all other shards are honest majority too. After completion of the first accept$(T, s_T)$ the shard removes the object from the active set, and thus subsequent $T'$ would also lead to pre-abort$(T, s_T)$. Thus there is no path in the chain of possible interleavings of the executions of two conflicting transactions that leads to them both being committed. □

**Security Theorem 8.** *(Validity) Under Honest Shards assumption, a transaction may only be accepted if it is valid according to the smart contract (or application) logic.*

*Proof.* A transaction is committed only if some nodes conclude that accept$(T, s_T)$, which presupposes all shards have provided enough evidence to conclude pre-accept$(T, s_T)$ for each of them. The concerned nodes include at least one shard per input object for the transaction; for any contract logic represented in the transaction, at least one of those shards will be managing object from that contract. Each shard checks the validity rules for the objects they manage (ensuring they are active) and the contracts those objects are part of (ensuring the transaction is valid with respect to the contract logic) in order to pre-accept$(T, s_T)$. Thus if all shards say pre-accept$(T, s_T)$ to conclude that accept$(T, s_T)$, all object have been checked as active, and all the contract calls within the transaction have been checked by at least one shard—whose decision is honest due to at most $f$ faulty nodes. If even a single object is

inactive or locked, or a single trace for a contract fails to check, then the honest nodes in the shard will emit pre-abort$(T, s_T)$, and the final decision will be abort$(T, s_T)$. □

## 4.7 Implementation & Evaluation

We implement a prototype of Byzcuit (Section 4.6) in Java and evaluate its performance and scalability. To analyze the overhead introduced by our replay attack defenses (*i.e.*, with message sequence numbers and dummy objects), we compare Byzcuit with replay defenses (`byzcuit`) with the baseline of Byzcuit without any replay attack defenses (`byzcuit-baseline`). Our implementation of Byzcuit is a fork of an early Chainspace code [25], and is released as an open-source project[10]. For BFT consensus, we use the BFT-SMaRt [143] Java library (based on PBFT [15]), which is one of the very few maintained open source BFT libraries. End users run a client to communicate with Byzcuit nodes, which sends transactions according to the BFT-SMaRt protocol. The Byzcuit client also acts as the Transaction Manager (TM) and is responsible for driving the cross-shard consensus. We evaluate the performance and scalability of our Byzcuit implementation through deployments on Amazon EC2 containers. We launch up to 96 instances for shard nodes and 96 instances for clients on *t2.medium* virtual machines, each containing 8 GB of RAM on 2 virtual CPUs and running GNU/Linux Debian 8.1. We use 4 nodes per shard. Each measured data point corresponds to 10 runs represented by error bars. The error bars in Figure 4.8 and Figure 4.9 show the average and standard deviation, and the error bars in Figure 4.10 show the median and the 75th and 25th percentiles.

**Throughput and scalability.** Figure 4.8 shows the throughput of Byzcuit (the number of transactions processed per second, tps) corresponding to an increasing number of shards. Each transaction has 2 input objects and 5 output objects, chosen randomly from shards. We test transactions with 5 output objects for a fair evaluation of Byzcuit's replay defenses by triggering the creation of dummy objects (*i.e.*, a large number of output objects and a small number of input objects implies a higher probability of output-only shards getting selected, triggering the creation of dummy

[10] `https://github.com/sheharbano/byzcuit`

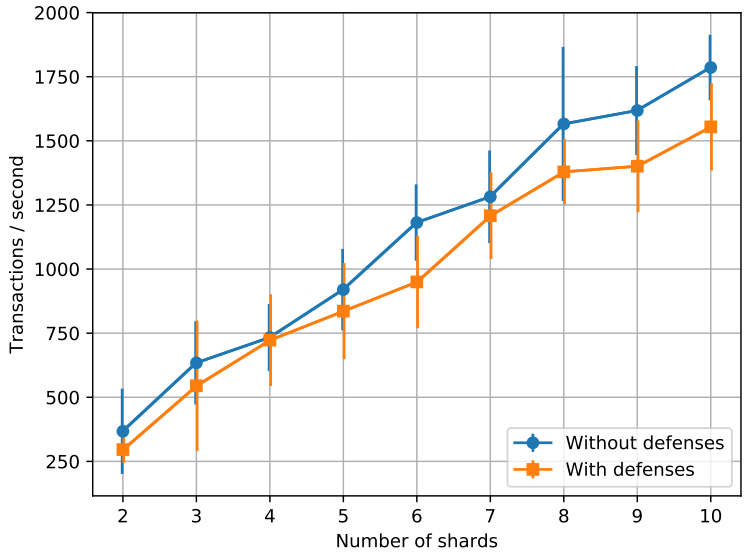


**Figure 4.8:** The effect of the number of shards on throughput. Each transaction has 2 input objects and 5 output objects, both chosen randomly from shards.

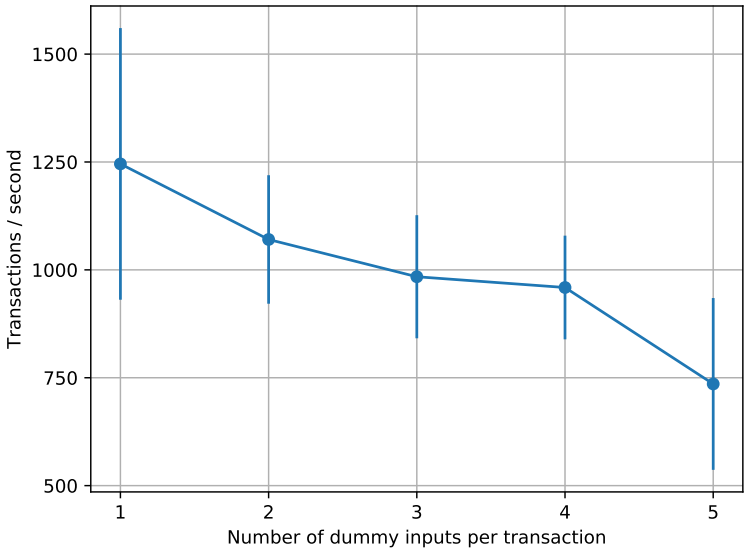


**Figure 4.9:** Decrease of Byzcuit throughput with the number of dummy objects. Each transaction has 1 input object, and up to 5 dummy objects randomly selected from unique non-input shards. 6 shards are used.

objects). We find that `byzcuit` has a throughput of 260 tps for 2 shards, and linearly scales with the addition of more shards achieving up to 1,550 tps for 10 shards. As expected, the throughput of `byzcuit` is lower than `byzcuit-baseline` by a somewhat constant factor ranging from 20–200 tps, but still increases linearly. This is expected because the creation of dummy objects in `byzcuit` leads to a higher number of shards processing the same transaction compared to `byzcuit-baseline`,

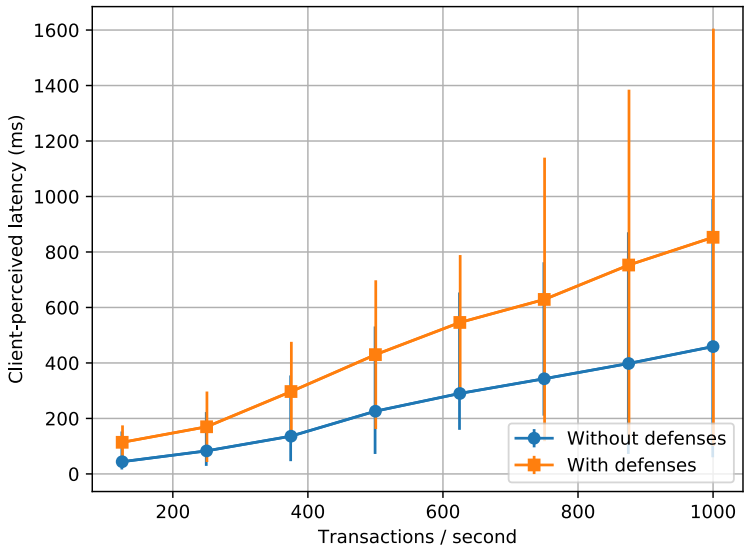


**Figure 4.10:** Client-perceived latency vs. system load (*i.e.* transactions received per second), for 6 shards with 2 inputs and 5 outputs per transaction (both chosen randomly from shards).

leading to lower concurrency and lower throughput.

**The effect of dummy objects on throughput.** We previously observed that dummy objects reduce the throughput of `byzcuit` with respect to `byzcuit-baseline`. Figure 4.9 shows the extent of throughput degradation due to dummy objects. We submit specially crafted transactions to 6 shards, such that each transaction has 1 input object, and we vary the number of dummy objects from 1–5 selected from unique shards, resulting in a corresponding decrease in concurrency because as many shards end up processing the transaction. For example, 2 dummy objects means that 3 shards process the transaction (1 input shard, and 2 more shards corresponding to the dummy objects). As expected, the throughput decreases by 20–250 tps with the addition of each dummy object, and reaches 750 tps when all 6 shards handle all transactions (which is the worst-case scenario).

**Client-perceived latency.** Figure 4.10 shows the client-perceived latency—the time from when a client submits a transaction, until it receives a decision from Byzcuit about whether the transaction has been committed—under varying system loads (expressed as transactions submitted to Byzcuit per second). We submit a total of 1,200 transactions at 200–1,000 transactions per second to Byzcuit with 6 shards. Each transaction has 2 inputs objects and 5 output objects, both chosen randomly

from shards. When the system is experiencing a load of up to 1,000 tps, clients hear back about their transactions in less than a second on average, even with our replay defenses in place.

## 4.8 Chapter Summary

This chapter presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend objects with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them. We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can be seen as unifying Atomix and S-BAC, into a protocol that is efficient and secure. We implemented and evaluated it on a real cloud-based testbed, showing that it can process over 1,550 tps for 10 shards, and that its capacity scales linearly with the number of shards.

# Chapter 5

# FastPay: High-Performance Byzantine Fault Tolerant Settlement

Chapters 3 and 4 demonstrate that sharded blockchains can be solid backbone systems that can scale to accommodate high throughput. However, their latency bottleneck remains their underlying consensus protocol which is too high for practical retail payments at physical points of sale. Chapter 4 shows that Byzcuit can process transactions in a few seconds but retail payments often require sub-second latency. This chapter overcomes this limitation through a side-infrastructure called FastPay; Figure 5.1 places FastPay in the big picture of this thesis. FastPay is a distributed settlement system for pre-funded payments that can be used as a financial side-infrastructure to support low-latency retail payments of a primary system such as Chainspace; it achieves extremely low latency by foregoing the expenses of consensus. The black diamonds in Figure 5.1 represent FastPay nodes and the white bidirectional arrow illustrates the communication between FastPay and Chainspace allowing users to move funds across those systems.

Settlement systems and Real-Time Gross Settlement Systems (RTGS) [144] constitute the most common approach to financial payments in closed banking networks, that is, between reputable institutions. In contrast, blockchain platforms have proposed a radically different paradigm, allowing account holders to interact directly with an online, yet highly secure, distributed ledger. Blockchain approaches aim to enable new use cases such as personal e-wallets or private transactions, and

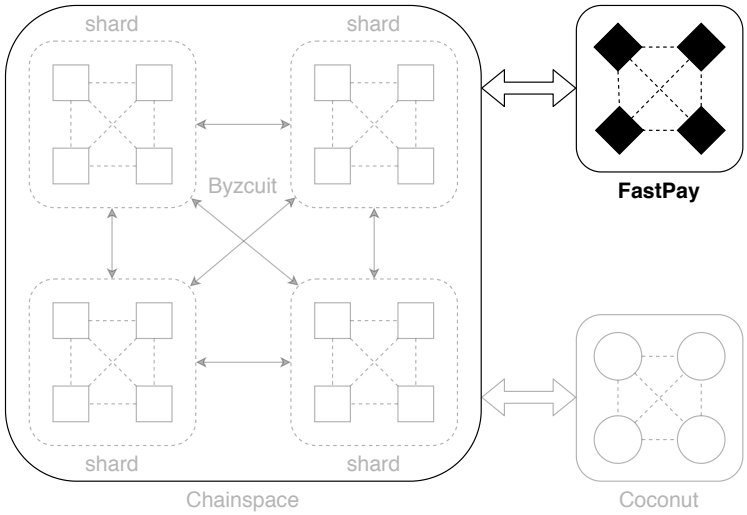


**Figure 5.1:** Global overview: FastPay.

generally provide ecosystems more favorable to users. However, until now, such open, distributed settlement solutions have come at a high performance cost and questionable scalability compared to traditional, closed RTGS systems.

FastPay is a Byzantine Fault Tolerant (BFT) real-time gross settlement (RTGS) system. It enables authorities to jointly maintain account balances and settle pre-funded retail payments between accounts. It supports extremely low-latency confirmation (sub-second) of eventual transaction finality, appropriate for physical point-of-sale payments. It also provides extremely high capacity, comparable with peak retail card network volumes, while ensuring gross settlement in real-time. FastPay eliminates counterparty and credit risks of net settlement and removes the need for intermediate banks, and complex financial contracts between them, to absorb these risks. FastPay can accommodate arbitrary capacities through efficient sharding architectures at each authority. Unlike any traditional RTGS, and more like permissioned blockchains, FastPay can tolerate up to $f$ Byzantine failures out of a total of $3f+1$ authorities, and retain both safety, liveness, and high-performance. FastPay can be deployed in a number of settings. First, it may be used as a settlement layer for a native token and crypto-currency, in a standalone fashion. Second, it may be deployed as a side-chain of another crypto-currency, or as a high performance settlement layer on the side of an established RTGS to settle fiat retail payments. In

this paper we present this second functionality in detail, since it exercises all features of the system, both payments between FastPay accounts, as well as payments into and out of the system.

## Contributions

This chapter makes the following contributions:

- The FastPay design is novel in that if forgoes full consensus; it leverages the semantics of payments to minimize shared state between accounts and to increase the concurrency of asynchronous operations; and it supports sharded authorities.
- We provide proofs of safety and liveness in a Byzantine and fully asynchronous network setting. We show that FastPay keeps all its properties despite the lack of total ordering, or asynchrony of updates to recipient accounts.
- We experimentally demonstrate comparatively very high throughput and low latency, as well as the robustness of the system under conditions of extremely high concurrency and load. We show that performance is maintained even when some (Byzantine) authorities fail.

## Outline

This paper is organized as follows: Section 5.2 introduces the entities within FastPay, their interactions, and the security properties and threat model. Section 5.3 details the design of FastPay both as a standalone system, and operated in conjunction with a Primary. Section 5.4 discusses safety and liveness. Section 5.5 briefly describes the implementation of the FastPay prototype. Section 5.6 provides a full performance evaluation of FastPay as we modulate its security parameters and load. Section 5.7 discusses key open issues such as privacy, governance mechanisms and economics of the platform. Section 5.8 covers the related work, both in terms of traditional financial systems and crypto-currencies. Section 5.9 concludes.

## 5.1 Background

Real-time gross settlement systems (RTGS) [144] are the backbone of modern financial systems. Commercial banks use them to maintain an account with central banks and settle large value payments. RTGS systems are limited in their capacity[1], making them unsuitable for settling low-value high-volume retail payments directly. Such retail payments are deferred: banks exchange information bilaterally about pending payments (often through SWIFT [146, 147]), they aggregate and net payments, and only settle net balances through an RTGS, often daily. The often quoted volume figure of around 80,000 transactions per second for retail card networks [148, 19] represents the rate at which 'promises' for payments are exchanged between banks, and not settled payments. Traditional RTGS systems are implemented as monolithic centralized services operated by a single authority, and must employ a number of technical and organizational internal controls to ensure they are reliable (through a primary-replica architecture with manual switch over) and correct—namely ensuring availability and integrity. Traditionally only regulated entities have accounts in those systems. This result in a Balkanized global financial system where financial institutions connect to multiple RTGS, directly or indirectly through corresponding banks, to execute international payments.

As mentioned in Chapter 2, blockchain-based technologies, starting with Bitcoin [17] in 2009, provide more open settlement systems often combined with their own digital tokens to represent value. Permissionless blockchains have been criticized for their low performance [21] in terms of capacity and finality times. However, a comparison with established settlement systems leads to a more nuanced assessment. Currently, Ethereum [24] can process 15 transactions per second. The actual average daily load on the EU ECB TARGET2 system is about 10 transactions per second [145] (in 2018) which is a comparable figure (and lower than the peak advertised capacity of 500 transaction per second). However, it falls very short of the advertised transaction rate of 80,000 transaction per second peak for retail payment networks—even though this figure does not represents settled transactions.

[1] For example, the relatively recent European Central Bank TARGET2 system has a maximum capacity of 500 transactions per second [145]

The stated ambitions of permissionless projects is to be able to settle transactions at this rate on an open and permissionless network, which remains an open research challenge [149]. Permissioned blockchains [150, 22] provide a degree of decentralization—allowing multiple authorities to jointly operate a ledger—at the cost of some off-chain governance to control who maintains the blockchain. The most prominent of such proposals is the Libra network [40], developed by the Libra association. Other technical efforts include Hyperledger [39], Corda [138] and Tendermint [62]. Those systems are based on traditional notions of Byzantine Fault Tolerant state machine replication (or sometimes consensus with crash-failures only), which presupposes an atomic commit channel (often referred as 'consensus') that sequences all transactions. Such architectures allow for higher capacities than Bitcoin and Ethereum. LibraBFT [41], for example, aims for 1,000 transactions per second at peak capacity [151, 152]; this exceeds many RTGS systems, but is still below the peak volumes for retail payment systems. A latency of multiple seconds when it comes to transaction finality is competitive with RTGS, but is unusable for retail payment at physical points of sale.

## 5.2 Overview

To illustrate its full capabilities, we describe FastPay as a side chain of a primary system holding the primary records of accounts. We call such a primary ledger *the Primary* for short, and its accounts *the Primary accounts*. The Primary can be instantiated as a programmable blockchain, through smart contracts, like Chainspace.

### 5.2.1 Participants

FastPay involves two types of participants: (i) authorities, and (ii) account owners (*users*, for short). All participants generate a key pair consisting of a private signature key and the corresponding public verification key. As a side-chain, FastPay requires a smart contract on the main blockchain, or a software component on an RTGS system that can authorize payments based on the signatures of a threshold of authorities from a *committee* with fixed membership.

By definition, an *honest* authority always follows the FastPay protocol, while a

*faulty* (or *Byzantine*) one may deviate arbitrarily. We present the FastPay protocol for $3f+1$ equally-trusted authorities, assuming a fixed (but unknown) subset of at most $f$ Byzantine authorities. In this setting, a *quorum* is defined as any subset of $2f+1$ authorities. (As for many BFT protocols, our proofs only use the classical properties of quorums thus apply to all Byzantine quorum systems [142].) When a protocol message is signed by a quorum of authorities, it is said to be *certified*: we call such a jointly signed message a *certificate*.

### 5.2.2 Accounts & Actions

A FastPay *account* is identified by its *address*, which we instantiate as the cryptographic hash of its public verification key. The state of a FastPay account is affected by four main high-level actions:

1. Receiving funds from a Primary account.
2. Transferring funds to a Primary account.
3. Receiving funds from a FastPay account.
4. Transferring funds to a FastPay account.

Actions (3) and (4) are necessary to execute payments within FastPay. Actions (1) and (2) are required only when interfacing with a primary system. FastPay also supports two read-only actions that are necessary to ensure liveness despite faults: reading the state of an account at a FastPay authority, and obtaining a certificate for any action executed by an authority.

### 5.2.3 Protocol Messages

The FastPay protocol consists of *transactions* on the Primary, denoted with letter $T$, and network requests that users send to FastPay authorities, which we call *orders* and denote with letter $O$. Users are responsible for broadcasting their orders to authorities and for processing the corresponding responses. The authorities are passive and *do not communicate directly with each other*.

**Transfer orders**. All transfers initiated by a FastPay account start with a *transfer order O* including the following fields:

- The sender's FastPay address, written $\texttt{sender}(O)$.
- The recipient, either a FastPay or a Primary address, written $\texttt{recipient}(O)$.
- A non-negative amount to transfer, written $\texttt{amount}(O)$.
- A sequence number $\texttt{sequence}(O)$.
- Optional user-provided data.
- A signature by the sender over the above data.

Authorities respond to valid transfer orders by counter-signing them (see Section 5.3 for validity checks). A quorum of such signatures is meant to be aggregated into a *transfer certificate*, noted $C$.

**Notations**. We write $O = \texttt{value}(C)$ for the original transfer order $O$ certified by $C$. For simplicity, we omit the operator `value` when the meaning is clear, *e.g.* $\texttt{sender}(C) = \texttt{sender}(\texttt{value}(C))$. FastPay addresses are denoted with letters $a$ and $b$. We use $\alpha$ for authorities and by extension for the shards of authorities.

### 5.2.4 Security Properties & Threat Model

FastPay guarantees the following security properties:

- **Safety:** No units of value are ever created or destroyed; they are only transferred between accounts.
- **Authenticity:** Only the owner of an account may transfer value out of the account.
- **Availability:** Correct users can always transfer funds from their account.
- **Redeemability:** A transfer to FastPay or Primary is guaranteed to eventually succeed whenever a valid transfer certificate has already been produced.
- **Public Auditability:** There is sufficient public cryptographic evidence for the state of FastPay to be audited for correctness by any party.

- **Worst-case Efficiency:** Byzantine authorities (or users) cannot significantly delay operations from correct users.

The above properties are maintained under a number of security assumptions: (i) there are at most $f$ Byzantine authorities out of $3f+1$ total authorities. (ii) The network is fully asynchronous, and the adversary may arbitrarily delay and reorder messages [45]. However, messages are eventually delivered. (iii) Users may behave arbitrarily but availability only holds for *correct users* (defined in Section 5.3.5). (iv) The Primary provides safety and liveness (when FastPay is used in conjunction with it). We further discuss the security properties of FastPay in Section 5.4.

## 5.3 The FastPay Protocol

FastPay authorities hold and persist the following information.

**Authorities**. The state of an authority $\alpha$ consists of the following information:

- The authority name, signature and verification keys.
- The committee, represented as a set of authorities and their verification keys.
- A map $\texttt{accounts}(\alpha)$ tracking the current account state of each FastPay address $a$ in use (see below).
- An integer value, noted $\texttt{last\_transaction}(\alpha)$, referring to the last transaction that paid funds into the Primary. This is used by authorities to synchronize FastPay accounts with funds from the Primary (see Section 5.3.3).

**FastPay accounts**. The state of a FastPay account $a$ within the authority $\alpha$ consists of the following:

- The public verification key of $a$, used to authenticate spending actions.
- An integer value representing the balance of payment, written $\texttt{balance}^a(\alpha)$.
- An integer value tracking the expected sequence number for the next spending action to be created, written $\texttt{next\_sequence}^a(\alpha)$. This value starts at 0 thus can be seen as the number of spending actions ever confirmed for this account.

- A field $\texttt{pending}^a(\alpha)$ tracking the last transfer order $O$ signed by $a$ such that the authority $\alpha$ considers $O$ as *pending confirmation*, if any; and absent otherwise.

- A list of certificates, written $\texttt{confirmed}^a(\alpha)$, tracking all the transfer certificates $C$ that have been *confirmed* by $\alpha$ and such that $\texttt{sender}(C) = a$. One such certificate is available for each sequence number $n$ ($0 \leq n < \texttt{next\_sequence}^a(\alpha)$).

- A list of *synchronization* orders, $\texttt{synchronized}^a(\alpha)$, having transferred funds from the Primary to account $a$. (See Section 5.3.3.)

We also define $\texttt{received}^a(\alpha)$ as the list of confirmed certificates for transfers received by $a$; formally:

$$\texttt{received}^a(\alpha) = \{C \text{ s.t. } \exists b.\, C \in \texttt{confirmed}^b(\alpha) \text{ and } \texttt{recipient}(C) = a\}$$

We assume arbitrary size integers. Although FastPay does not let users overspend, (temporary) negative balances for account states are allowed for technical reasons discussed in Section 5.4. When present, a pending (signed) transfer order $O = \texttt{pending}^a(\alpha)$ effectively locks the sequence number of the account $a$ and prevents $\alpha$ from accepting new transfer orders from $a$ until *confirmation*, that is, until a valid transfer certificate $C$ such that $\texttt{value}(C) = O$ is received. This mechanism can be seen as the 'Signed Echo Broadcast' implementation of a Byzantine consistent broadcast on the label (account, next sequence number) [38].

**Storage considerations**. The information contained in the lists of certificates $\texttt{confirmed}^a(\alpha)$ and $\texttt{received}^a(\alpha)$ and in the synchronization orders $\texttt{synchronized}^a(\alpha)$ is self-authenticated—being signed by a quorum of authorities and by the Primary, respectively. Remarkably, this means that authorities may safely outsource these lists to an external high-availability data store. Therefore, FastPay authorities only require a constant amount of local storage per account, rather than a linear amount in the number of transactions.

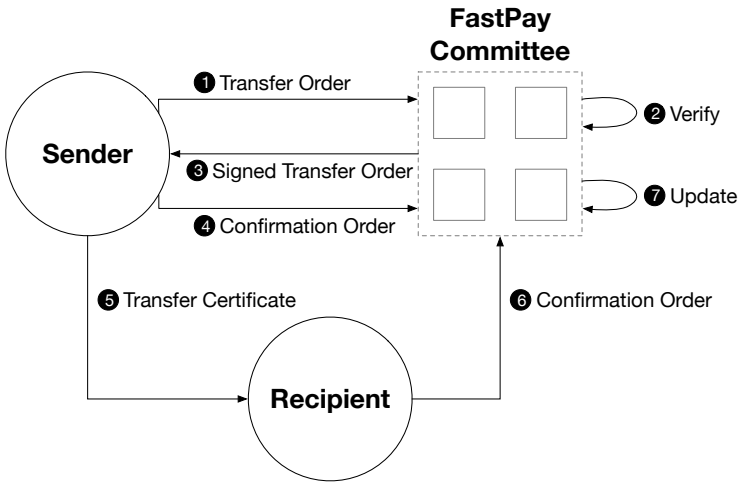


**Figure 5.2:** Transfer of funds from FastPay to FastPay.

### 5.3.1 Transferring Funds within FastPay

FastPay operates by implementing a Byzantine consistent broadcast channel per account, specifically using a 'Signed Echo Broadcast' variant (Algorithm 3.17 in [38]). It operates in two phases and all messages are relayed by the initiating user. Consistent Broadcast ensures *Validity*, *No duplication*, *Integrity*, and *Consistency*. It always terminates when initiated by a correct user. However, if a FastPay user equivocates, current operations may fail, and the funds present on the users account may become inaccessible.

**Transferring funds**. Figure 5.2 illustrates a transfer of funds within FastPay. To transfers funds to another FastPay account, the sender creates a FastPay transfer order ($O$) with the next sequence number in their account, and signs it. They then send the FastPay transfer order to all authorities. Each authority checks (❶) (i) that the signature is valid, (ii) that no previous transfer is pending (or its for the same transfer), (iii) that the amount is positive, (iv) that the sequence number matches the expected next one ($\texttt{sequence}(O) = \texttt{next\_sequence}^a(\alpha)$), and (iv) that the balance ($\texttt{balance}^a(\alpha)$) is sufficient (❷). Then, it records the new transfer as pending and sends back a signature on the transfer order (❸) which is also stored. The authority algorithm to handle transfer orders, corresponding to step ❷, is presented in Figure 5.3. The user collects the signatures from a quorum of authorities, and use them along the FastPay transfer order to form a transfer

certificate. The sender provides this transfer certificate to the recipient as a proof that the payment will proceed (❺). To conclude the transaction, the sender (❹) or the recipient (❻) must broadcast the transfer certificate ($C$) to the authorities (we call this a *confirmation order*)[2]. Upon reception of a confirmation order for the current sequence number, each authority $\alpha$ (❼) (i) checks that a quorum of signatures was reached, (ii) decreases the balance of the sender, (iii) increments the sequence number ($\texttt{next\_sequence}^a(\alpha)+1$) to ensure 'deliver once' semantics, and (iv) sets the pending order to None ($\texttt{pending}^a(\alpha) = \text{None}$). Each authority $\alpha$ also (v) adds the certificate to the list $\texttt{confirmed}^a(\alpha)$, and (vi) increases the balance of the recipient account asynchronously (*i.e.* without sequencing this write in relation to any specific payments from this account across authorities). The authority algorithm to handle confirmation orders (as in step ❼) is presented in Figure 5.3.

In Section 5.4, we show that the FastPay protocol is safe thanks to the semantics of payments into an account and their commutative properties. FastPay is a significant simplification and deviation from an orthodox application of Guerraroui *et al.* [153], where accounts are single-writer objects and all write actions are mediated by the account owner. FastPay allows payments to be executed after a single consistent broadcast, rather than requiring recipients to sequence payments into their accounts separately. This reduces both latency and the state necessary to prevent replays.

**Payment finality**. Once a transfer certificate *could* be formed, namely $2f+1$ authorities signed a transfer order, no other order can be processed for an account until the corresponding confirmation order is submitted. Technically, the payment is final: it cannot be canceled, and will proceed eventually. As a result, showing a transfer certificate to a recipient convinces them that the payment will proceed. We call showing a transfer certificate to a recipient a *confirmation*, and then subsequently submitting the confirmation order, to move funds, *settlement*. *Confirmation* requires only a single round trip to a quorum of authorities resulting in very low-latency (see Section 5.6), and giving the system its name.

[2]Aggregating signed transfer orders into a transfer certificate does not requires knowledge of any secret; therefore, anyone (and not only the sender or the recipient) can broadcast the transfer certificate to the authorities to conclude the transaction.

```
fn handle_transfer_order(α, O) -> Result {
    /// Check shard and signature.
    ensure!(α.in_shard(sender(O)));
    ensure!(O.has_valid_signature());

    /// Obtain sender account.
    match accounts(α).get(sender(O)) {
        None => bail!(),
        Some(account) => {
            /// Check if the same order is already pending.
            if let Some(pending) = account.pending {
                ensure!(pending.transfer == O);
                return Ok();
            }
            ensure!(account.next_sequence == sequence(O));
            ensure!(account.balance >= amount(O));
            /// Sign and store new transfer.
            account.pending = Some(α.sign(O));
            return Ok();
}   }   }

fn handle_confirmation_order(α, C)
    -> Result<Option<CrossShardUpdate>> {
    /// Check shard and certificate.
    ensure!(α.in_shard(sender(C)));
    ensure!(C.is_valid(α.committee));
    let O = value(C);

    /// Obtain sender account.
    let sender_account =
        accounts(α).get(sender(O))
        .or_insert(AccountState::new());

    /// Ignore old certificates.
    if sender_account.next_sequence > sequence(O) {
        return Ok(None);
    }

    /// Check sequence number and balance.
    ensure!(sender_account.next_sequence == sequence(O));
    ensure!(sender_account.balance >= amount(O));

    /// Update sender account.
    sender_account.balance -= amount(O);
    sender_account.next_sequence += 1;
    sender_account.pending = None;
    sender_account.confirmed.push(C);

    /// Update recipient locally or cross-shard.
    let recipient = match recipient(O) {
        Address::FastPay(recipient) => recipient,
        Address::Primary(_) => { return Ok(None) }
    };

    /// Same shard: read and update the recipient.
    if α.in_shard(recipient) {
        let recipient_account = accounts(α).get(recipient)
            .or_insert(AccountState::new());
        recipient_account.balance += amount(O);
        return Ok(None);
    }

    /// Other shard: request a cross-shard update.
    let update = CrossShardUpdate {
        shard_id: α.which_shard(recipient),
        transfer_certificate: C,
    };
    Ok(Some(update))
}
```

**Figure 5.3:** Authority algorithms for handling transfer and confirmation orders. (The cross-shard update logic is in presented in Figure 5.4.)

```
fn handle_cross_shard_commit(α, C) -> Result {
    let O = value(C);
    let recipient = match recipient(O) {
        Address::FastPay(recipient) => recipient,
        Address::Primary(_) => { bail!(); }
    };
    ensure!(α.in_shard(recipient));
    let recipient_account = accounts(α).get(recipient)
        .or_insert(AccountState::new());
    recipient_account.balance += amount(O);
    Ok()
}

fn handle_primary_synchronization_order(α, S) -> Result {
    /// Update recipient(S) assuming that S comes from
    /// a trusted source (e.g. Primary client).
    let recipient = recipient(S);
    ensure!(α.in_shard(recipient));

    if transaction_index(S) <= last_transaction(α) {
        /// Ignore old synchronization orders.
        return Ok();
    }
    ensure!(transaction_index(S) == last_transaction(α) + 1);

    last_transaction(α) += 1;
    let recipient_account = accounts(α).get(recipient)
        .or_insert(AccountState::new());
    recipient_account.balance += amount(S);
    recipient_account.synchronized.push(S);
    Ok()
}
```

**Figure 5.4:** Authority algorithms for cross-shard updates and (Primary) synchronization orders.

**Proxies, gateways and crash recovery**. The protocols as presented involve the sender being on-line and mediating all communications. However, the only action that the sender *must* perform personally is forming a transfer order, requiring their signature. All subsequent operations, including sending the transfer order to the authorities, forming a certificate, and submitting a confirmation order can be securely off-loaded to a proxy trusted only for liveness. Alternatively, a transfer order may be given to a merchant (or payment gateway) that drives the protocol to conclusion. In fact, any party in possession of a signed transfer order may attempt to make a payment progress concurrently. And as long as the sender is correct the protocol will conclude (and if not may only lock the account of the faulty sender).

This provides significant deployment and implementation flexibility. A sender client may be implemented in hardware (in a NFC smart card) that only signs transfer orders. These are then provided to a gateway that drive the rest of the protocol. Once the transfer order is signed and handed over to the gateway the sender may go off-line

or crash. Authorities may also attempt to complete the protocol upon receiving a valid transfer order. Finally, the protocol recovers from user crash failures: anyone may request a transfer order that is partially confirmed from any authority, proceed to form a certificate, and submit a confirmation order to complete the protocol.

### 5.3.2 Sharding authorities

FastPay requires minimal state sharing between accounts, and allows for a very efficient sharding at each authority by account. The consistent broadcast channel is executed on a per-account basis. Therefore, the protocol does not require any state sharing between accounts (and shards) up to the point where a valid confirmation order has to be settled to transfer funds between FastPay accounts. On settlement, the sender account is decremented and the funds are deposited into the account of the recipient, requiring interaction between at most two shards (second algorithm of Figure 5.3). Paying into an account can be performed asynchronously, and is an operation that cannot fail (if the account does not exist it is created on the spot). Therefore, the shard managing the recipient account only needs to be notified of the confirmed payment through a reliable, deliver once, authenticated, point to point channel (that can be implemented using a message authentication code, inter-shard sequence number, re-transmission, and acknowledgments) from the sender shard. This is a greatly simplified variant of a two-phase commit protocol coordinated by the sender shard (for details see the Presume Nothing and Last Agent Commit optimizations [154, 155]). Modifying the validity condition of the consistent broadcast to ensure the recipient account exists (or any other precondition on the recipient account) would require a full two-phase commit before an authority signs a transfer order, and can be implemented while still allowing for (slightly less) efficient sharding. The algorithms in fig. 5.3 implement shading. An authority shard checks that the transfer order ($O$) or certificate ($C$) is to be handled by a specific shard and otherwise rejects it without mutating its state. Handling confirmation orders depends on whether a recipient account is on the same shard. If so, the recipient account is updated locally. Otherwise, a *cross shard message* is created for the recipient shard to update the account (see code in the Appendix for this operation).

The ability to shard each authority has profound implications: increasing the number of shards at each authority, increases the theoretical throughput linearly, while latency remains constant. Our experimental evaluation confirms this experimentally (see Section 5.6).

### 5.3.3 Interfacing with the Primary

We describe the protocols required to couple FastPay with the Primary, namely transferring funds from the Primary to a FastPay account, and conversely from a FastPay to a Primary account. We refer throughout to the logic on the Primary as a *smart contract*, and the primary store of information as the *blockchain*. A traditional RTGS would record this state and manage it in conventional ways using databases and stored procedures, rather than a blockchain and smart contracts. We write $\sigma$ for the state of the 'blockchain' at a given time, and $\texttt{transactions}(\sigma)$ for the set of FastPay transactions $T$ already processed by the blockchain.

**Smart contract**. The smart contract mediating interactions with the Primary requires the following data to be persisted in the blockchain:

- The FastPay committee composition: a set of authority names and their verification keys.
- A map of accounts where each FastPay address is mapped to its current Primary state (see below).
- The total balance of funds in the smart contract, written $\texttt{total\_balance}(\sigma)$.
- The transaction index of the last transaction that added funds to the smart contract, written $\texttt{last\_transaction}(\sigma)$.

**Accounts**. The Primary state of a FastPay account $a$ consists of the set of sequence numbers of transfers already executed from this account to the Primary. This set is called the *redeem log* of $a$ and written $\texttt{redeemed}^a(\sigma)$.

**Adding funds from the Primary to FastPay**. Figure 5.5 shows a transfer of funds from the Primary to FastPay. The owner of the FastPay account (or anyone else)

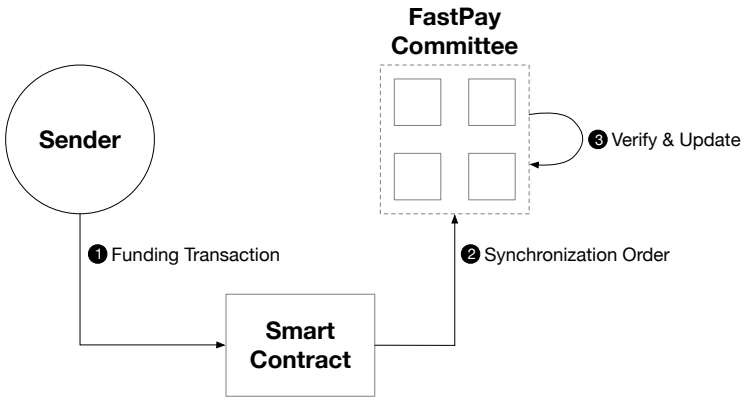


**Figure 5.5:** Transfer of funds from the Primary to FastPay.

starts by sending a payment to the FastPay smart contract using a Primary transaction (❶). This transaction is called a *funding transaction*, and includes the recipient FastPay address for the funds and the amount of value to transfer. When the Primary transaction is executed, the FastPay smart contract generates a *Primary event* that instructs authorities of a change in the state of the FastPay smart contract. We assume that each authority runs a full Primary client to authenticate such events. For simplicity, we model such an event as a *(Primary) synchronization order* (❷). The smart contract ensures this event and the synchronization order contain a unique, always increasing, sequential transaction index. When receiving a synchronization order, each authority (i) checks the transaction index follows the previously recorded one, (ii) increments the last transaction index in their global state, (iii) creates a new FastPay account if needed, and (iv) increases the account balance of the target account by the amount of value specified (❸). Figure 5.4 presents the authority algorithm for handling funding transactions.

**Transferring funds from FastPay to the Primary**. Figure 5.6 shows a transfer of funds from FastPay to Primary. The FastPay sender signs a *Primary transfer order* using their account key and broadcasts it to the authorities (❶). This is simply a transfer order with a Primary address as the recipient. Once a quorum of signatures is reached (❷ and ❸), the sender creates a certified (Primary) transfer order, also called a *transfer certificate* for short. The sender broadcasts this certificate to authorities to confirm the transaction (❹) and unlock future spending from this account. When

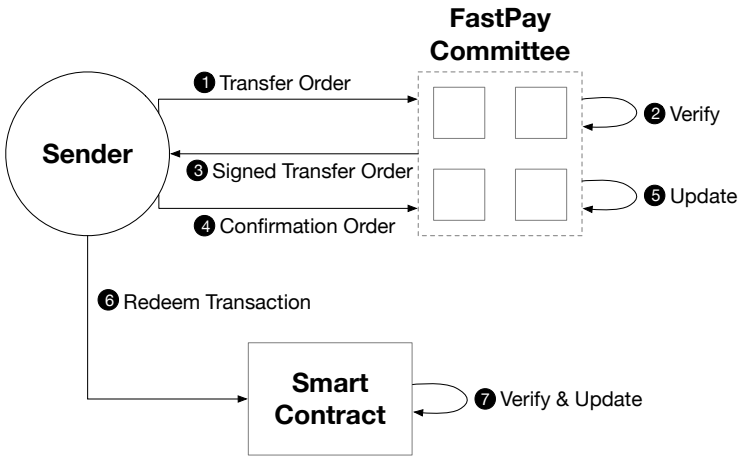


**Figure 5.6:** Transfer of funds from FastPay to the Primary.

an authority receives a confirmation order containing a certificate of transfer (❺), it must (i) check that a quorum of signatures was reached, and (ii) that the account sequence number matches the expected one; (iii) they then set the pending order to None, (iv) increment the sequence number, and (v) decrease the account balance. Finally, the recipient of the transfer should send a redeem transaction to the FastPay smart contract on the Primary blockchain (❻). When the FastPay smart contract receives a valid redeem transaction (❼), it must (i) check that the sequence number is not in the Primary redeem log of the sender, to prevent reuse; (ii) update this redeem log; (iv) transfer the amount of value specified from the smart contract into the recipient's Primary account.

### 5.3.4 State Recovery & Auditing

For every account $a$, each authority $\alpha$ must make available the pending order $\texttt{pending}^a(\alpha)$, the sequence number $\texttt{next\_sequence}^a(\alpha)$, the synchronization orders $\texttt{synchronized}^a(\alpha)$, and the certificates confirmed so far, indexed by senders (*i.e.* $\texttt{confirmed}^a(\alpha)$) and receivers ($\texttt{received}^a(\alpha)$). Sharing these data fulfills two important roles: (i) this lets anyone read the state of any incomplete transfer and drive the protocol all the way to settlement; (ii) it enables auditing authority states and detect Byzantine faults (*e.g.* incorrect balance checks).

### 5.3.5 Correct Users & Client Implementation

A *correct user* owning a FastPay account $a$ follows the correctness rules below:

1. The user sets the sequence number of a new transfer order $O$ to be the next expected integer after the previous transfer (starting with 0); *i.e.* they sign exactly one transfer order per sequence number;

2. They broadcast the new transfer order $O$ to enough authorities until they (eventually) obtain a certificate $C$;

3. They successfully broadcast the certificate $C$ to a quorum of authorities.

**FastPay Client**. To address the correctness rules above, our reference implementation of a FastPay client holds and persists the following minimal state:

- The address $a$ and the secret key of the account;
- The FastPay committee;
- The sequence number to be used in the next transfer;
- The transfer order that it signed last, in case it is still pending.

In this setting, the available balance of a user account is not tracked explicitly but rather evaluated (conservatively) from the Primary transactions and the available logs for incoming transfers and outgoing transfers (Section 5.3.4). Evaluating the balance before starting a transfer is recommended, as signing a transfer order with an excessive amount will block (correct) client implementations from initiating further transfers until the desired amount is available.

## 5.4 Security Analysis

This section discusses safety and liveness of FastPay. Let $\sigma$ denote the current state of the Primary. We define $\texttt{funding}^a(\sigma)$ as the sum of all the amounts transferred to

a FastPay address $a$ from the Primary:

$$\mathtt{funding}^a(\sigma) = \sum_{\begin{cases} T \in \mathtt{transactions}(\sigma) \\ \mathtt{recipient}(T) = a \end{cases}} \mathtt{amount}(T)$$

For simplicity, we write $\sum_C \mathtt{amount}(C)$ when we mean to sum over certified transfer orders: $\sum_{O \text{ s.t. } \exists C.O=\mathtt{value}(C)} \mathtt{amount}(O)$. We define $\mathtt{funding}^a(\alpha)$ as the sum of all the amounts received from the Primary by a FastPay address $a$, as seen at a given time by an authority $\alpha$:

$$\mathtt{funding}^a(\alpha) = \sum_{S \in \mathtt{synchronized}^a(\alpha)} \mathtt{amount}(S)$$

### 5.4.1 Safety

**Lemma 2** (Transfer certificate uniqueness)**.** *If*

$$\begin{cases} \mathtt{sender}(C) = \mathtt{sender}(C'), \text{ and} \\ \mathtt{sequence}(C) = \mathtt{sequence}(C') \end{cases}$$

*then $C$ and $C'$ certify the same transfer order:*

$$\mathtt{value}(C) = \mathtt{value}(C')$$

*Proof.* Both certificates $C$ and $C'$ are signed by a quorum of authorities. By construction, any two quorums intersect on at least one honest authority. Let $\alpha$ be an honest authority in both quorums. $\alpha$ signs at most one transfer order per sequence number, thus $C$ and $C'$ certify the same transfer order. □

**Lemma 3** (FastPay invariant)**.** *For every honest authority $\alpha$, for every account $a$, it holds that*

$$\left(\mathtt{balance}^a(\alpha) + \sum_{C \in \mathtt{confirmed}^a(\alpha)} \mathtt{amount}(C)\right) \leq \left(\mathtt{funding}^a(\alpha) + \sum_{C \in \mathtt{received}^a(\alpha)} \mathtt{amount}(C)\right)$$

*Besides, if* $n = \texttt{next\_sequence}^a(\alpha)$*, we have that* $\texttt{confirmed}^a(\alpha) = \{C_0 \dots C_{n-1}\}$ *for some certificates* $C_k$ *such that* $\texttt{sequence}(C_k) = k$ *and* $\texttt{sender}(C_k) = a$*.*

*Proof.* By construction of the FastPay authorities (Figure 5.3 and Figure 5.4): Whenever a confirmed certificate $C$ is added to $\texttt{confirmed}^a(\alpha)$, $\texttt{balance}^a(\alpha)$ is decreased by $\texttt{amount}(C)$, and $\texttt{next\_sequence}^a(\alpha)$ is incremented into $\texttt{sequence}(C)+1$. Any new synchronization order equally increases $\texttt{balance}^a(\alpha)$ and $\texttt{funding}^a(\alpha)$. Whenever a confirmed certificate $C$ is added to $\texttt{received}^a(\alpha)$, possibly later (due to cross-shard updates), $\texttt{balance}^a(\alpha)$ is increased once by $\texttt{amount}(C)$. □

**Lemma 4** (Primary invariant)**.** *The total balance of all FastPay accounts on Primary is such that*

$$\texttt{total\_balance}(\sigma) = \left( \sum_a \texttt{funding}^a(\sigma) - \sum_{C \in \texttt{redeemed}(\sigma)} \texttt{amount}(C) \right)$$

*Proof.* By construction of the smart contract handling funding and redeeming transactions (Section 5.3.3): whenever a funding transaction $T$ is executed by the smart contract, both $\texttt{funding}^a(\sigma)$ and $\texttt{total\_balance}(\sigma)$ increase by $\texttt{amount}(T)$. Conversely, $\texttt{total\_balance}(\sigma)$ decreases by $\texttt{amount}(C)$ whenever a Primary transfer certificate $C$ is redeemed on-chain and added to $\texttt{redeemed}(\sigma)$. □

**Lemma 5** (Funding log synchronization)**.** *For every honest authority* $\alpha$ *and every account a, it holds that*

$$\texttt{funding}^a(\alpha) \leq \texttt{funding}^a(\sigma)$$

*Proof.* By definition of the synchronization with the Primary (Section 5.3.3), and by security of the Primary and its client, $\texttt{funding}^a(\alpha)$ only increases after a funding transaction has already increased $\texttt{funding}^a(\sigma)$ by the same amount. □

**Lemma 6** (Balance check)**.** *For every honest authority* $\alpha$*, whenever* $O = \texttt{pending}^a(\alpha)$ *holds, then we also have*

$$\texttt{amount}(O) \leq \texttt{balance}^a(\alpha)$$

*Proof.* By construction of the FastPay authorities (Figure 5.3), if $O = \texttt{pending}^a(\alpha)$, then $O$ was successfully processed by $\alpha$ as a new transfer order from account $a$. At the time of the request, $\texttt{amount}(O)$ did not exceed the current balance $B$. Since $O$ is still pending, in the meantime, no other transfer certificates from account $a$ have been confirmed by $\alpha$. (A confirmation would reset the field `pending` and prevent $O$ from being pending again due to the increased sequence number.) Therefore, the balance did not decrease, and $\texttt{balance}^a(\alpha) \geq B \geq \texttt{amount}(O)$. □

**Proposition 1** (Account safety)**.** *For every account a, at any given time, we have that*

$$\sum_{\texttt{sender}(C)=a} \texttt{amount}(C) \;\leq$$

$$\texttt{funding}^a(\sigma) \;+\; \sum_{\texttt{recipient}(C)=a} \texttt{amount}(C)$$

*Proof.* Let $n$ be the highest sequence number of a transfer certificate $C_n$ from $a$. Let $\alpha$ an honest authority whose signature is included in the certificate. At the time of signature, we had $\texttt{value}(C_n) = \texttt{pending}^a(\alpha)$, therefore by Lemma 3 and Lemma 6:

$$\left(\texttt{amount}(C_n) \;+\; \sum_{C\in\texttt{confirmed}^a(\alpha)} \texttt{amount}(C)\right)$$
$$\leq \left(\texttt{funding}^a(\alpha) \;+\; \sum_{C\in\texttt{received}^a(\alpha)} \texttt{amount}(C)\right)$$

Given that $n$ is the highest sequence number, by Lemma 2 and Lemma 3, the left-hand term exactly covers the certified transfer orders from $a$ and is equal to $\sum_{\texttt{sender}(C)=a} \texttt{amount}(C)$.

Given that amounts are non-negative, for every honest node $\alpha$, we have

$$\sum_{C\in\texttt{received}^a(\alpha)} \texttt{amount}(C) \;\leq \sum_{\texttt{recipient}(C)=a} \texttt{amount}(C)$$

Finally, $\texttt{funding}^a(\alpha) \leq \texttt{funding}^a(\sigma)$ by Lemma 5. □

**Security Theorem 9** (Solvency of FastPay)**.** *At any time, the sum of the amounts of all existing certified transfers from FastPay to the Primary cannot exceed the funds collected by all transactions on the Primary smart contract:*

$$\sum_{\texttt{recipient}(C)\in\texttt{Primary}} \texttt{amount}(C) \;\leq \sum_{a} \texttt{funding}^a(\sigma)$$

*Proof.* By applying Proposition 1 on every account and summing, we obtain:

$$\begin{aligned}
&\textstyle\sum_a \texttt{funding}^a(\sigma) \geq \\
&\quad \left( \textstyle\sum_C \texttt{amount}(C) - \sum_{\texttt{recipient}(C) \in \texttt{FastPay}} \texttt{amount}(C) \right) \\
&\quad = \textstyle\sum_{\texttt{recipient}(C) \in \texttt{Primary}} \texttt{amount}(C)
\end{aligned}$$

□

### 5.4.2 Liveness

Next, we describe how receivers of valid transfer certificates can finalize transactions and make funds available on their own accounts (Primary and FastPay).

**Proposition 2** (Redeemability of valid transfer certificates to Primary)**.** *A new valid Primary transfer certificate C can always be redeemed by sending a new redeem transaction T to the smart contract.*

*Proof.* Theorem 9 shows that the smart contact always has enough funding for all certified Primary transfer orders. The definition of $\texttt{redeemed}(\sigma)$ (Section 5.3.3) thus ensures that any new certified Primary transfer order can be redeemed exactly once. □

**Proposition 3** (Redeemability of valid transfer certificates to FastPay)**.** *Any user can eventually have a valid FastPay transfer certificate C confirmed by any honest authority.*

*Proof.* If a certificate $C$ exists for account $a$ and sequence number $n$, this means at least $f+1$ honest authorities contributed signatures to the transfer order $O = \texttt{value}(C)$. By construction of FastPay, these authorities have received (Figure 5.3) and will keep available (Section 5.3.4) all the previous confirmation orders $C_0 \ldots C_{n-1}$ with $\texttt{sender}(C_k) = a$, $\texttt{sequence}(C_k) = k$. Therefore, any client can retrieve them and eventually bring any other honest authority up to date with $C$. □

Specifically, in Proposition 3, the confirmation order for $C$ is guaranteed to succeed for every honest authority $\alpha$, provided that the user first recovers and transfers to $\alpha$ all the *missing certificates required by* $\alpha$, defined as the sequence $C_k \ldots C_{n-1}$ such that $k = \texttt{next\_sequence}^a(\alpha)$, $a = \texttt{sender}(C)$, $n = \texttt{sequence}(C)$,

$\texttt{sender}(C_i) = a$ $(k \leq i \leq n-1)$. The fact that no other certificates need to be confirmed (*e.g.* to credit the balance of $\texttt{sender}(C)$ itself) is closely related to the possibility of (temporary) negative balances for authorities.

Note that having a FastPay certificate confirmed by an authority $\alpha$ only affects $\alpha$'s recipient and the sender's balances (*i.e. redeems the certificate*) the first time it is confirmed. Finally, we state that FastPay funds credited on an account can always be spent. We write $\texttt{received}(a)$ for the set of *incoming* transfer certificates $C$ such that $\texttt{recipient}(C) = a$ and $C$ is known to the owner of the account $a$.

**Proposition 4** (Availability of transfer certificates)**.** *Let $a$ be an account owned by a correct user, $n$ be the next available sequence number after the last signed transfer order (if any, otherwise $n = 0$), and $O$ be a new transfer order signed by $a$ with $\texttt{sequence}(O) = n$ and $\texttt{sender}(O) = a$.*

*Assume that the owner of $a$ has secured enough funds for a new order $O$ based on their knowledge of the chain $\sigma$, the history of outgoing transfers, and the set $\texttt{received}(a)$. That is, formally:*

$$\left(\texttt{amount}(O) + \sum_{\substack{\texttt{sender}(C) = a \\ \texttt{sequence}(C) < n}} \texttt{amount}(C)\right) \leq \left(\texttt{funding}^a(\sigma) + \sum_{C \in \texttt{received}(a)} \texttt{amount}(C)\right)$$

*Then, for any honest authority $\alpha$, the user will always eventually obtain a valid signature of $O$ from $\alpha$ after sending the following orders to $\alpha$:*

1. *A synchronization order from the Primary based on the known state $\sigma$;*
2. *A confirmation order for every $C \in \texttt{received}(a)$, preceded by all the missing certificates required by $\alpha$ (if any) for the sender of $C$;*
3. *Then, the transfer order $O$.*

*Proof.* Let $B \geq \texttt{amount}(O)$ be the following value evaluated at the time of the creation of the new transfer order $O$:

$$B = \left( \mathtt{funding}^a(\sigma) - \sum_{\substack{\mathtt{sender}(C) = a \\ \mathtt{sequence}(C) < n}} \mathtt{amount}(C) + \sum_{C \in \mathtt{received}(a)} \mathtt{amount}(C) \right)$$

By a case analysis similar to the proof of Lemma 3, provided that the owner of $a$ is communicating the information described in Proposition 4 to the authority $\alpha$, it will hold eventually that $\mathtt{balance}^a(\alpha) \geq B \geq \mathtt{amount}(O)$ and $\mathtt{next\_sequence}^a(\alpha) = n$. We deduce that eventually $\alpha$ will accept the transfer order $O$ and make the value of its signed (pending) order available. □

### 5.4.3 Performance under Byzantine Failures

The FastPay protocol does not rely on any designated leader (like PBFT [15]) to make progress or create proposals; FastPay authorities do not directly communicate with each other, and their actions are symmetric. Clients create certificates by gathering the first $2f+1$ responses to a valid transfer order, and no action of a Byzantine authority may delay the creation of a certificate. A Byzantine authority may not even present a signature on a different order as a response to confuse a correct client, since it would have to be signed by the correct payer. Subsequently, a correct client submits the confirmation order to all authorities. Again, Byzantine authorities cannot in any way delay honest authorities from processing the payment locally in their databases, and enabling a subsequent payment for the sending account.

Byzantine clients may attempt denial of service attacks by over-using the system, and for example creating a very large number of receiving accounts (this could be disincentivized by charging some fee for an account creation). However, an attempt to equivocate by sending two transfer orders for a single sequence number could either result in their own account being locked (no single transfer order can achieve $2f+1$ signatures to form a certificate and move to the next sequence number), or one of them succeeding—neither of which degrade performance. Transfer orders with insufficient funds or incorrect sequence numbers are simply rejected, which does not significantly affect performance (if anything they do not result in confirmation orders that are more costly to process than transfer orders, see Section 5.6).

### 5.4.4 Worst-Case Efficiency of FastPay Clients

To initiate a transfer (Proposition 4) or receive funds (Proposition 3) from a sender account $a$, a FastPay client must address a quorum of authorities. During the exchange, each authority $\alpha$ may require missing certificates $C_k \ldots C_{n-1}$, where $k = \texttt{next\_sequence}^a(\alpha)$ is provided by $\alpha$. In an attempt to slow down the client, a Byzantine authority could return $k = 0$ and/or fail to respond at some point. To address this, a client should query each authority $\alpha$ in parallel. After retrieving the sequence number $k$, the required missing certificates should be downloaded sequentially, in reverse order, then forwarded to $\alpha$. Given that FastPay client operations succeed as soon as a quorum of authorities complete their exchanges, this strategy ensures client efficiency despite Byzantine authorities.

## 5.5 Implementation

We implemented both a FastPay client and a networked multi-core multi-shard FastPay authority in Rust, using Tokio[3] for networking and ed25519-dalek[4] for signatures. For the verification of the multiple signatures composing a certificate we use ed25519 batch verification. To reduce latency we use UDP for FastPay requests and replies, and make the core of FastPay idempotent to tolerate retries in case of packet loss; we also provide an experimental FastPay implementation using exclusively TCP. Currently, data-structures are held in memory rather than persistent storage. We implement an authority shard as a separate operating system process with its own networking and Tokio reactor core, to validate the low overhead of intra-shard coordination (through message passing rather than shared memory). We experimented with manually pinning processes to physical cores without a noticeable increases in performance through the Linux *taskset* feature. It seems the Linux OS does a good job of distributing processes and keeping them on inactive cores. We also experimented with a single process multi-threaded implementation of FastPay, using a single Tokio reactor for all shards on multi-core machines. However, this led to significantly lower performance, and therefore we opted for using separate

[3] `https://tokio.rs`
[4] `https://github.com/dalek-cryptography/ed25519-dalek`

processes even on a single machine for each shard. The exact bottleneck justifying this lower performance—whether at the level of Tokio multi-threading or OS resource management—still eludes us.

The implementation for both server and client is less than 4,000 LOC (of which half are for the networking), and a further 1,375 LOC of unit tests. It required about 2.5 months of work for 3 engineers, and a bit over 1,500 git commits. Keeping the core small required constant re-factoring and its simplicity is a significant advantage of the proposed FastPay design. We are open sourcing the Rust implementation, Amazon web services orchestration scripts, benchmarking scripts, and measurements data to enable reproducible results[5].

## 5.6 Evaluation

We evaluate the throughput and latency of our implementation of FastPay through experiments on AWS. We particularly aim to demonstrate that (i) sharding is effective, in that it increases throughput linearly as expected; (ii) latency is not overly affected by the number of authorities or shards, and remains near-constant, even when some authorities fail; and (iii) that the system is robust under extremely high concurrency and transaction loads.

### 5.6.1 Microbenchmarks

We report on microbenchmarks of the single-CPU core time required to process transfer orders, authority signed partial certificates, and certificates. Table 5.1 displays the cost of each operation in micro seconds ($\mu s$) assuming 10 authorities (recall $1\mu s = 10^{-6}s$); each measurement is the result of 500 runs on an Apple laptop (MacBook Pro) with a 2.9 GHz Intel Core i9 (6 physical and 12 logical cores), and 32 GB 2400 MHz DDR4 RAM. The first 3 rows respectively indicate the time to create and serialize (i) a transfer order, (ii) a partial certificate signed by a single authority, and (iii) a transfer certificate as part of a confirmation order. The last 3 rows indicate the time to deserialize them and check their validity. The dominant CPU cost involves the deserialization and signature check on certificates ($236\mu s$), which

[5] `https://github.com/calibra/fastpay`

| Measure | Mean ($\mu$s) | Std. ($\mu$s) |
|---|---|---|
| Create & Serialize Order | 27 | 1 |
| Create & Serialize Partial Cert. | 27 | 2 |
| Create & Serialize Certificate | 4 | 0 |
| Deserialize & Check Order | 58 | 1 |
| Deserialize & Check Partial Cert. | 60 | 1 |
| Deserialize & Check Certificate | 236 | 10 |

**Table 5.1:** Microbenchmark of single core CPU costs of FastPay operations; average and standard deviation of 500 measurements for 10 authorities.

includes the batch verification of the 8 signatures (7 from authorities and 1 from sender). However, deserializing orders ($58\mu s$) and votes ($60\mu s$) is also expensive: it involves 1 signature verification (no batching) and creating 1 signature. Those results indicate that a single core shard implementation may only settle just over 4,000 transactions per second—highlighting the importance of sharding to achieve high-throughput. In terms of networking costs, a transfer order is 146 bytes, and the signed response is 293 bytes. This could be reduced by only responding with a signature (64 bytes) rather than the full signed order, but we chose to echo back the order to simplify client implementations. A full certificate for an order is 819 bytes, and the response—consisting of an update on the state of the FastPay account—is 51 bytes. For deployments using many authorities we can compress certificates by using an aggregate signature scheme (such as BLS [113]). However, verification CPU costs of BLS only make this competitive for committees larger than 50-100 authorities. We note that all FastPay message types fit within the common maximum transmission unit of commodity IP networks, allowing requests and replies to be executed using a single UDP packet (assuming no packets loss and 10 authorities).

### 5.6.2 Throughput

We deploy a FastPay multi-shard authority on Amazon Web Services (Stockholm, eu-north-1 zone), on a m5d.metal instance. This class of instance guarantees 96 virtual CPUs (48 physical cores), on a 2.5 GHz, Intel Xeon Platinum 8175, and 384 GB memory. The operating system is Linux Ubuntu server 18.04, where we increase

the network buffer to about 96MB. In all graphs, each measurement is the average of 9 runs, and the error bars represent one standard deviation; all experiments use our UDP implementation. We measure the variation of throughput with the number of shards. Our baseline experiment parameters are: 4 authorities (for confirmation orders), a load of 1M transactions, and applying back-pressure to allow a maximum of 1000 concurrent transactions at the time into the system (*i.e.* the *in-flight* parameter). We then vary those baseline parameters through our experiments to illustrate their impact on performance.

**Robustness and performance under high concurrency**. Figures 5.7 and 5.8 respectively show the variation of the throughput of processing transfer and confirmation orders as we increase the number of shards per authority, from 15 to 85. We measure those by processing 1M transactions, across 4 authorities. Figure 5.7 shows that the throughout of transfer orders slowly increases with the number of shards. The in-flight parameter—the maximum number of transactions that is allowed into the system at any time—influences the throughput by about 10%, and setting it to 1,000 seems optimal for performance. The degree of concurrency in a system depends on the number of concurrent client requests, and we observe that FastPay is stable and performant even under extremely high concurrency peaks of 50,000 concurrent requests. Afterwards, the Operating System UDP network buffers fill up, and the authority network stacks simply drop the requests.

Figure 5.8 shows that the throughput of confirmation orders initially increases linearly with the number of shards, and then reaches a plateau at around 48 shards. This happens because our experiments are run on machines with 48 physical cores, running at full speed, and 48 logical cores. The in-flight parameter of concurrent requests does not influence the throughput much, but setting it too low (*e.g.* at 100) does not saturate our CPUs. These figures show that FastPay can support up to 160,000 transactions per second on 48 shards (about 7x the peak transaction rate of the Visa payments network [19]) while running on commodity computers that cost less than 4,000 USD/month per authority[6].

[6] AWS reports a price of 5.424 USD/hour for their `m5d.metal` instances. `https://aws.amazon.com/ec2/pricing/on-demand` (January 2020)

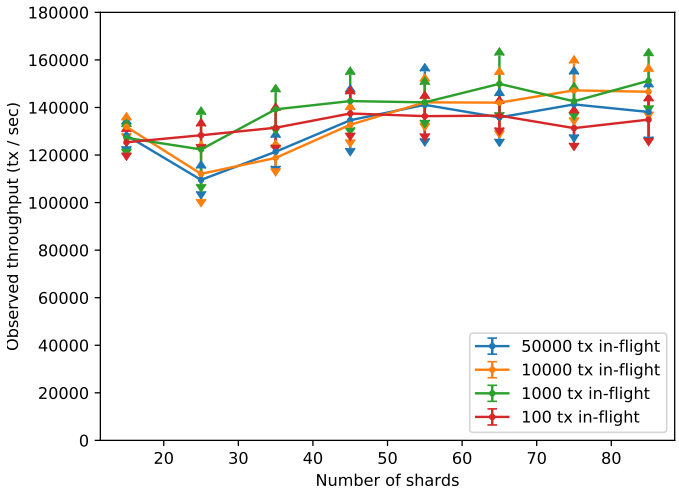


**Figure 5.7:** Variation of the throughput of transfer orders with the number of shards, for various levels of concurrency (in-flight parameter). The measurements are run under a total load of 1M transactions.

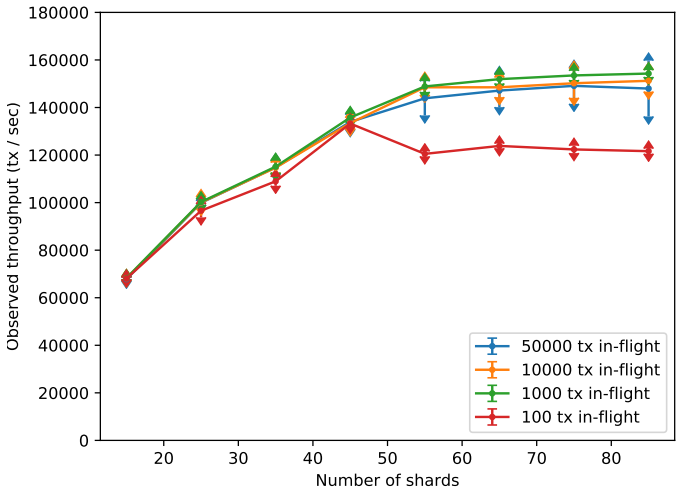


**Figure 5.8:** Variation of the throughput of confirmation orders with the number of shards, for various levels of concurrency (in-flight parameter). The certificates are issued by 4 authorities, and the measurements are run under a total load of 1M transactions.

**Robustness and performance under total system load**. Figures 5.9 and 5.10 show the variation of the throughput of transfer and confirmation orders with the number of shards, for various total system loads—namely the total number of transactions in the test; they show that the throughput is not affected by the system load. The tests were performed with 4 authorities, and the client concurrency in-flight parameter set

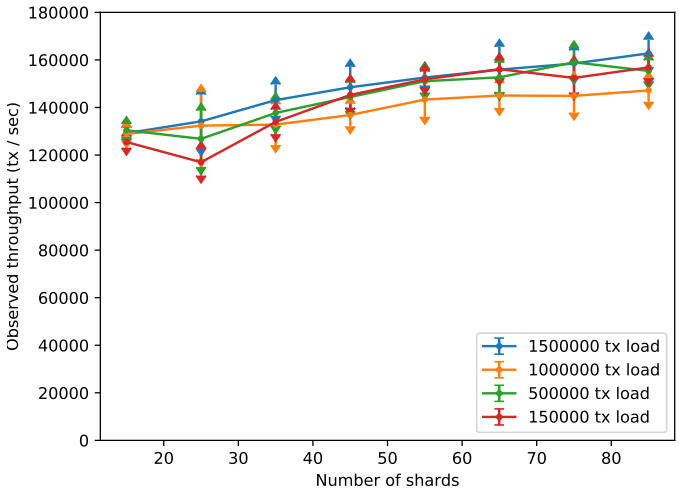


**Figure 5.9:** Variation of the throughput of transfer orders with the number of shards, for various loads. The in-flight parameter is set to 1,000.

to 1,000. These figures illustrate that FastPay can process about 160,000 transactions per second even under a total load of 1.5M transactions, and that the total load does not significantly affect performance. These supplement Figures 5.7 and 5.8 that illustrate that the concurrent transaction rate also does not influence performance significantly (except when it is too low by under-utilizing the system).

Readers may be surprised those measurements are important. The key measurement work by Han *et al.* [156] compares a number of permissioned systems under a high load, and shows that for all of Hyperledger Fabric (v0.6 with PBFT) [157], Hyperledger Fabric (v1.0 with BFT-Smart) [158], Ripple [159] and R3 Corda v3.2 [160] the successful requests per second *drops to zero* as the transaction rate increases to more than a few thousands transactions per second (notably for Corda only a few hundred). An exception is Tendermint [62], that maintains a processed transaction rate of about 4,000 to 6,000 transactions per second at a high concurrency rate. Those findings were confirmed for Hyperledger Fabric that reportedly starts saturating at a rate of 10,000 transactions per second [161]. Our results demonstrate that FastPay continues to be very performant even under the influence of extremely high rates of concurrent transactions (in-flight parameter) and overall work load (total number of transactions processed), as expected. This is apparently not the norm.

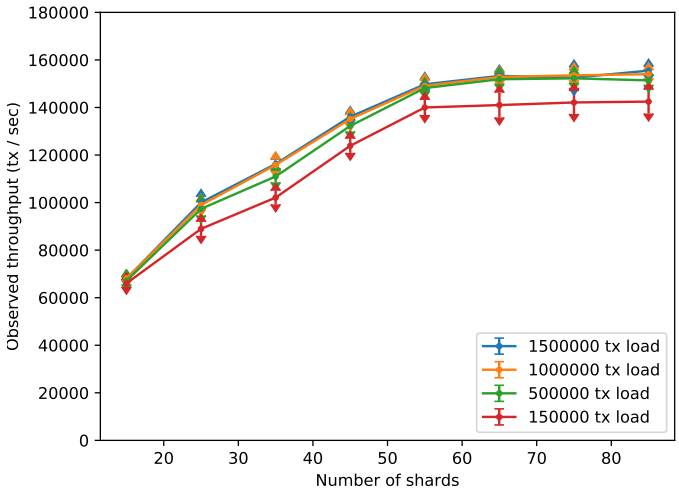


**Figure 5.10:** Variation of the throughput of confirmation orders with the number of shards, for various loads. The certificates are issued by 4 authorities, and the in-flight parameter is set to 1,000.

**Influence of the number of authorities**. As discussed in Section 5.3, we expect that increasing the number of authorities only impacts the throughput of confirmation orders (that need to transfer and check transfer certificates signed by $2f+1$ authorities), and not the throughput of transfer orders. Figure 5.11 confirms that the the throughput of confirmation orders decreases as the number of authorities increases. FastPay can still process about 80,000 transactions per second with 20 authorities (for 75 shards). The measurements are taken with an in-flight concurrency parameter set to 1,000, and under a load of 1M total transactions. We note that for higher number of authorities, using an aggregate signature scheme (*e.g.* BLS [113]) would be preferable since it would result in constant time verification and near-constant size certificates. However, due to the use of batch verification of signatures, the break even point may be after 100 authorities in terms of verification time.

### 5.6.3 Latency

We measure the variation of the client-perceived latency with the number of authorities. We deploy several FastPay multi-shard authorities on Amazon Web Services (all in Stockholm, eu-north-1 zone), each on a m5d.8xlarge instance. This class of instance guarantees 10Gbit network capacity, on a 3.1 GHz, Intel Xeon Platinum

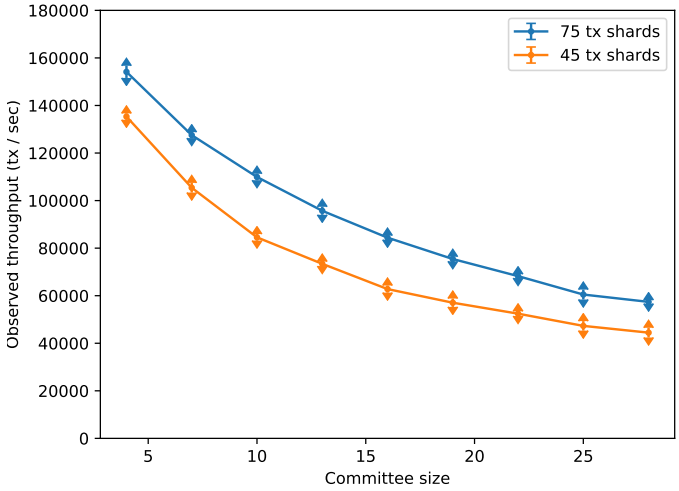


**Figure 5.11:** Variation of the throughput of confirmation orders with the number of authorities, for various number of shards. The in-flight parameter is set to 1,000 and the system load is of 1M transactions.

8175 with 32 cores, and 128 GB memory. The operating system is Linux Ubuntu server 16.04. Each instance is configured to run 15 shards. The client is run on an Apple laptop (MacBook Pro) with a 2.9 GHz Intel Core i9 (6 physical and 12 logical cores), and 32 GB 2400 MHz DDR4 RAM; and connected to a reliable WIFI network. We run experiments with the client in two different locations; (i) in the U.K. (geographically close to the authorities, same continent), and (ii) in the U.S. West Coast (geographically far from the authorities, different continent). Each measurement is the average of 300 runs, and the error bars represent one standard deviation; all experiments use our UDP implementation.

We observe that the client-authority WAN latency is low for both transfer and confirmation orders; the latency is under 200ms when the client is in the U.S. West Coast, and about 50ms when the client is in the U.K. Figure 5.12 illustrates the latency between a client creating and sending a transfer order to all authorities, and receiving sufficient signatures to form a transfer certificate (in our experiment we wait for all authorities to reply to measure the worse case where $f$ authorities are Byzantine). The latency is virtually constant as we increase the number of authorities, due to the client emitting orders asynchronously to all authorities and waiting for responses in parallel. Figure 5.13 illustrates the latency to submit a

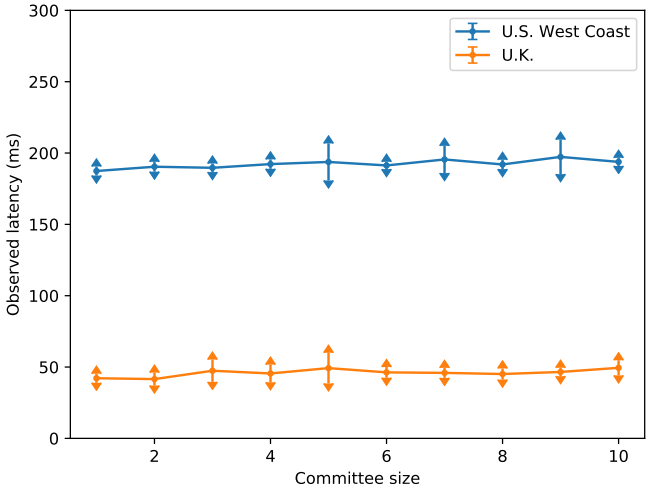


**Figure 5.12:** Variation of the latency of transfer orders with the number of authorities, for various locations of the client.

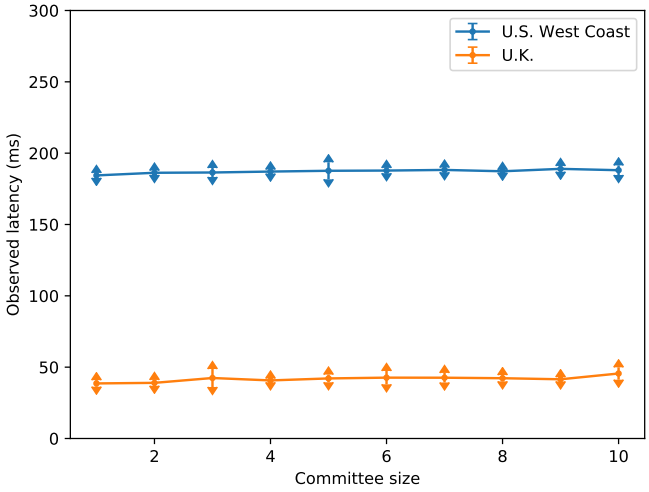


**Figure 5.13:** Variation of the latency of confirmation orders with the number of authorities, for various locations of the client.

confirmation order, and wait for all authorities to respond with a success message. It shows latency is virtually constant when increasing the number of authorities. This indicates that the latency is largely dominated by the network (and not by the verification of certificates). However, since even for 10 authorities a FastPay message fits within a network MTU, the variation is very small. Due to our choice of using UDP as a transport there is no connection initiation delay (as for TCP), but we may

observe packet loss under very high congestion conditions. Authority commands are idempotent to allow clients to re-transmit to overcome loss without sacrificing safety.

**Performance under failures**. Research literature suggests permissioned blockchains based on (often leader-based) consensus suffer an enormous performance drop when some authorities fail [162]. We measure the effect of authority failure in FastPay and show that latency is not affected when $f$ or fewer authorities are unavailable. We run our baseline experimental setup (10 authorities distributed over 10 different AWS instances), when a different number of authorities are not available for $f = 0 \ldots 3$. We measure the latency experienced by a client on the same continent (Europe), sending a transfer order until it forms a valid transfer certificate. Table 5.2 summarizes the mean latency and standard deviation for different $f$. There is no statistically significant difference in latency, no matter how many tolerable failures FastPay experiences (up to $f \leq 3$ for 10 authorities). We also experimented with killing authorities one by one with similar results, up to $f > 3$ when the system did observably lose liveness as expected. The underlying reason for the steady performance under failures is FastPay's lack of reliance on a leader to drive the protocol.

| $f$ | Mean (ms) | Std. (ms) |
|---|---|---|
| 0 | 43 | 2 |
| 1 | 41 | 3 |
| 2 | 44 | 4 |
| 3 | 47 | 2 |

**Table 5.2:** Crash-failure Latency.

## 5.7 Limitations & Future Work

**Threats to validity of experiments**. Our experiments represent the best case performance, for a set number of authorities and shards, as they are performed in laboratory conditions. In particular, real-wold transactions may have the same sender account, which would prevent them from being executed in parallel. Further, the throughput evaluation places transaction load on an authority through the local network interface, and therefore does not take fully into account the operating system networking costs of a full WAN stack. Further, our WAN latency experiments were performed against

authorities with very low-load. Finally, the costs of persisting databases to storage are not taken into account when measuring latency and throughput (we leave the implementation of low-latency persistent storage to future work).

**Checkpointing, authority, and key rotation**. The important enabler for the good performance of FastPay, but also an important limitation, is the fact that authorities do not need to reach consensus on the state of their databases. We demonstrate that payments are secure in this context, but various system maintenance operations are harder to implement. For example, checkpointing the state of all accounts in the systems, to compress the list of stored certificates would be beneficial, but cannot be straightforwardly implemented without consensus. Similarly, it would be beneficial for authorities to be able to rotate in and out of the committee, as well as to update their cryptographic signature keys. Due to the lack of tight synchronization between authorities there is no natural point that guarantees they all update their committees at the same logical time. Further, our proofs of liveness under asynchrony presume that transfer orders and certificates that were once valid, will always be valid. Integrating such governance features into FastPay will require careful design to safely leverage either some timing (synchrony) assumptions or use a more capable (but maybe lower performance) consensus layer, such as one facilitated by the Primary.

**Economics and fees**. Some cost to insert transactions into a system (like fees in Bitcoin), allows for sound accounting and prevents Denial of Service attacks by clients over-using an open system. The horizontal scalability of FastPay alleviates somehow the need to integrate such a scheme, since issues of capacity can be resolved by increasing its capacity through more shards (as well as deploying network level defenses). However, if there was a need to implement fees for using FastPay we would not recommend using micro-payments associated with each payment like in Bitcoin. Rather, we would recommend allowing a client to deposit some payment into a service account with all authorities, and then allow them to deduct locally some of this fee for any services rendered (namely any signed transfer order or confirmation order processed). In practical terms, the variable costs of processing transactions in FastPay is low. There is no artificial shortage due to lack of scalability,

and a flat periodic fee on either senders or recipients might be sufficient to support operations (rather than a charge per transaction).

## 5.8 Comparison with Related Works

We compare FastPay with traditional payment systems and some relevant crypto-currencies based on permissioned blockchains.

**Traditional payment systems**. In the context of traditional payment systems FastPay is a real-time gross settlement system (RTGS) [163, 144]—payments are executed in close to real-time, there is no netting between participants, and the transfer of funds is final upon the full payment protocol terminating. All payments are pre-funded so there is no need to keep track of credit or liquidity, which makes the design vastly simpler. A well known issue with RTGS systems is the need for higher liquidity, as compared with settlement systems based on daily settlement after netting—since more money moves around exposing accounts to higher volatility. In that respect FastPay is state of the art, in that it allows immediate liquidity recycling [164], namely as soon as a payment is processed the value payed into an account may be used to pay other parties in the system.

FastPay, from an assurance and performance perspective is significantly superior to deployed RTGS systems: it (i) implements a fully Byzantine fault tolerant architecture (established systems rely on master-slave configurations to only recover from few crash failures), (ii) has higher throughput (as compared, for example with the TARGET2 [145] European Central Bank RTGS systems that has a target throughput of 500 tx/sec), and (iii) faster finality (as compared to TARGET2 providing finality of a few seconds). Since FastPay allows for fast gross settlement, participants are not exposed to credit risk, as is the case for retail payment systems such as VISA and Mastercard (that use daily netting, and have complex financial arrangements to mitigate credit risk in case of bank default). Furthermore, it does achieve both throughput and latency, comparable to those systems combined—about 80,000 tx/sec at peak times, when adding up the throughput of Visa and Mastercard together [19, 148]. On the downside, FastPay lacks certain features of mature RTGS

systems: in particular it does not support Delivery-on-Payment transactions that atomically swap securities when payment is provided, or Payment-versus-Payment, that atomically swap amounts in different currencies to minimize the risk of foreign exchange transactions. These require atomic operations across accounts controlled by different users, and would therefore require extending FastPay to support them (namely operations with consensus number of 2 per Herlihy [165]).

**Crypto-currencies**. FastPay provides high assurance in the context of Byzantine failures within its infrastructure. So in that respect it is comparable with systems encountered in the space of permissioned blockchains and crypto-currencies, as well as their eco-system of payment channels. FastPay is permissioned in that the set of authorities managing the system is closed—in fact we do not even propose a way to rotate those authorities and leave this to future work. Qualitatively, FastPay differs from other permissioned (or permissionless) crypto-currencies in a number of important ways: it is secure under full network asynchrony (since it does not require or rely on atomic broadcast channels or consensus, but only consistent broadcast)—leading to higher performance. This direction was explored in the past in relation to central bank crypto-currency systems [91] and high performance permissionless systems [46]. It was recently put on a formal footing by Guerraroui *et al.* [153]. Our work extends this theory to allow increased concurrency, correctness under sharding, and rigorous interfacing with external settlement mechanisms. FastPay achieves auditability through a set of certificates signed by authorities rather than a sequential log of actions (blockchain), which would require authorities to reach agreement on a common sequence. Quantitatively, compared with other permissioned systems FastPay is extremely performant. HyperLedger Fabric [39] running with 10 nodes achieves about 1,000 transactions per second and a latency of about 10 seconds [161]; and Libra [40] and Corda [138, 166] achieve similar performance. JP Morgan developed a digital coin built from the Ethereum codebase, which can achieve about 1,500 transactions per second with four nodes, and imposing a block time of 1 second [167]. Tendermint [62] reportedly achieves 10,000 transactions per second with 4 nodes, with a few seconds latency [168]. However, as we discussed in

Section 5.6, many of those systems see their performance degrading dramatically under heavy load—whereas FastPay performs as expected.

FastPay can be used as a side chain of any crypto-currency with reasonable finality guarantees, and sufficient programmability. As compared to bilateral payment channels it is superior in that it allows users to pay anyone in the system without locking liquidity into the bilateral channel, and is fully asynchronous. However, FastPay does rely on an assumption of threshold non-Byzantine authorities for safely and liveness, whereas payment channel designs only rely on network synchrony for safety and liveness (under conditions of asynchrony safety may be lost). As compared to payment channel networks (such as the lighting network [169]) FastPay is simpler and does not require complex path finding algorithms [169, 170, 171, 172].

## 5.9 Chapter Summary

FastPay is a settlement layer based on consistent broadcast channels, rather than full consensus. The FastPay design leverages the nature of payments to allow for asynchronous payments into accounts, and optional interactions with an external Primary to build a practical system, while providing proofs of both safely and liveness; it also proposes and evaluates a design for sharded implementation of authorities to horizontally scale and match any throughput need. The performance and robustness of FastPay is beyond and above the state of the art, and validates that moving away from both centralized solutions and full consensus to manage pre-funded retail payments has significant advantages. Authorities can jointly process tens of thousands of transactions per second (we observed a peak of 160,000 tx/sec) using merely commodity hardware and lean software. A payment confirmation latency of less than 200ms between continents make FastPay practical for point of sale payments—where goods and services need to be delivered fast and in person. Pretty much instant settlement enables retail payments to be freed from intermediaries, such as banks payment networks, since they eliminate any credit risk inherent in deferred netted end-of-day payments, that underpin today most national Fast Payment systems [173]. Further, FastPay can tolerate up to one-third of authorities crashing or even becom-

ing Byzantine without losing either safety or liveness (or performance). This is in sharp contrast with existing centralized settlement layers operating on specialized mainframes with a primary / backup crash fail strategy (and no documented technical strategy to handle Byzantine operators). Surprisingly, it is also in contrast with permissioned blockchains, which have not achieved similar levels of performance and robustness yet, due to the complexity of engineering and scaling full byzantine fault-tolerant consensus protocols.

# Chapter 6

# Coconut: Threshold Issuance Selective Disclosure Credentials

Chainspace (Chapter 3) is the backbone of this thesis. It provides a sharded blockchain that can scale to accommodate high throughput and can process transactions in a few seconds by running Byzcuit at its core (Chapter 4). Further, it revisits the execution of smart contracts on blockchains by introducing a model where transactions are executed at the client side to support privacy-preserving applications by design. This last chapter presents Coconut, the final component of this thesis that reveals the full potential of Chainspace's execution model by presenting a number of decentralised and scalable privacy-persevering applications implemented as Chainspace smart contracts. Coconut is selective disclosure credential scheme that natively integrates with blockchains; Figure 6.1 places Coconut in the big picture of this thesis by instantiating it as a side-infrastructure of Chainspace. The black circles in Figure 6.1 represent Coconut nodes and the white bidirectional arrow illustrates the communication between Coconut and Chainspace allowing users to obtain and use Coconut credentials within Chainspace smart contracts.

Selective disclosure credentials [107, 108] allow the issuance of a credential to a user, and the subsequent unlinkable revelation (or 'showing') of some of the attributes it encodes to a verifier for the purposes of authentication, authorization or to implement electronic cash. However, as explained in Section 2.5.2, established schemes have shortcomings. Some entrust a single issuer with the credential sig-

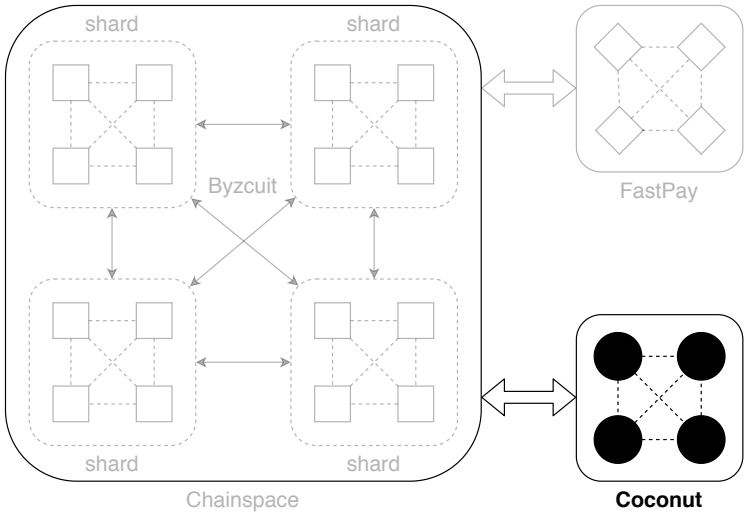


**Figure 6.1:** Global overview: Coconut.

nature key, allowing a malicious issuer to forge any credential or electronic coin. Other schemes do not provide the necessary efficiency, re-randomization, or blind issuance properties necessary to implement practical selective disclosure credentials. No existing scheme provides all of efficiency, threshold distributed issuance, private attributes, re-randomization, and unlinkable multi-show selective disclosure. This is especially troublesome in applications related to e-cash or token schemes, since the issuer is effectively given a license to generate coins, and the unlinkable nature of the showing protocols prevents auditors from detecting such behaviour. Moreover, the security models of these systems generally assume that integrity should hold in the presence of a threshold number of dishonest or faulty nodes (Byzantine fault tolerance); it is desirable for similar assumptions to hold for multiple credential issuers (threshold issuance). Integrity should hold when a threshold of infrastructure nodes are honest; however, not all authorities are expected to be online or honest. The lack of efficient general purpose selective disclosure credentials impacts platforms that support 'smart contracts', such as Ethereum [24], Hyperledger [39] and Chainspace [25]. They all share the limitation that verifiable smart contracts may only perform operations recorded on a public blockchain. Thus, smart contracts themselves cannot execute operations requiring secrets, such as issuing signatures or credentials. Chainspace overcomes this limitation by providing an infrastructure

where smart contracts are partially executed client-side, and thus allowing a framework where they can operate on the users' secret inputs. However, the nodes backing the ledger can only operate on public data, which prevents smart contracts from directly accessing any node-side secret input.

Issuing credentials through smart contracts would be very desirable: a smart contract could conditionally issue user credentials depending on the state of the blockchain, or attest some claim about a user operating through the contract—such as their identity, attributes, or even the balance of their wallet. This is not possible, as current selective credential schemes would either entrust a single party as an issuer, or would not provide appropriate efficiency, re-randomization, blind issuance and selective disclosure capabilities (as in the case of threshold signatures [174]). For example, the Hyperledger system supports CL credentials [107] through a trusted third party issuer, illustrating their usefulness, but also their fragility against the issuer becoming malicious.

Coconut addresses these challenges, and allows a subset of decentralized mutually distrusting authorities to jointly issue credentials, on public or private attributes. Those credentials cannot be forged by users, or any small subset of potentially corrupt authorities. Credentials can be re-randomized before selected attributes are shown to a verifier, protecting privacy even in the case in which all authorities and verifiers collude. The Coconut scheme is based on a threshold issuance signature scheme that allows partial claims to be aggregated into a single credential. Mapped to the context of permissioned and semi-permissioned blockchains, Coconut allows collections of authorities in charge of maintaining a blockchain, or a side chain [174] based on a federated peg, to jointly issue selective disclosure credentials. Coconut uses short and computationally efficient credentials, and efficient revelation of selected attributes and verification protocols. Each partial credential and the consolidated credential is composed of exactly two group elements. The size of the credential remains constant regardless of the number of attributes or authorities/issuers. Furthermore, after a one-time setup phase where the users collect and aggregate a threshold number of verification keys from the authorities, the attribute showing and verification are $O(1)$

in terms of both cryptographic computations and communication of cryptographic material—irrespective of the number of authorities. Our evaluation of the Coconut primitives shows very promising results. Verification takes about 10ms, while signing a private attribute is about 3 times faster. The latency is about 600 ms when the client aggregates partial credentials from 10 authorities distributed across the world.

## Contributions

This chapter makes the following key contributions:

- It describes the signature schemes underlying Coconut, including how key generation, distributed issuance, aggregation and verification of signatures operate. The scheme is an extension and hybrid of the Waters signature scheme [114], the BGLS signature [116], and the signature scheme of Pointcheval and Sanders [109]. This is the first general purpose, fully distributed threshold issuance, multi-show credential scheme of which we are aware.

- It uses Coconut to implement a generic smart contract library for Chainspace [25] and one for Ethereum [24], performing public and private attribute issuance, aggregation, randomization and selective disclosure. We evaluate their performance and cost within those platforms.

- It presents the design of three applications using the Coconut contract library: a coin tumbler providing payment anonymity; a privacy preserving electronic petitions; and a proxy distribution system for a censorship resistance system. We implement and evaluate the first two applications on the Chainspace platform, and provide a security and performance evaluation.

## Outline

Section 6.1 presents an overview of the Coconut system; Section 6.2 presents the cryptographic primitives underlying Coconut; Section 6.4 and Section 6.5 respectively present an implementation and some example of applications backed by the Coconut system. Section 6.6 provides the evaluation of the core primitives and of the

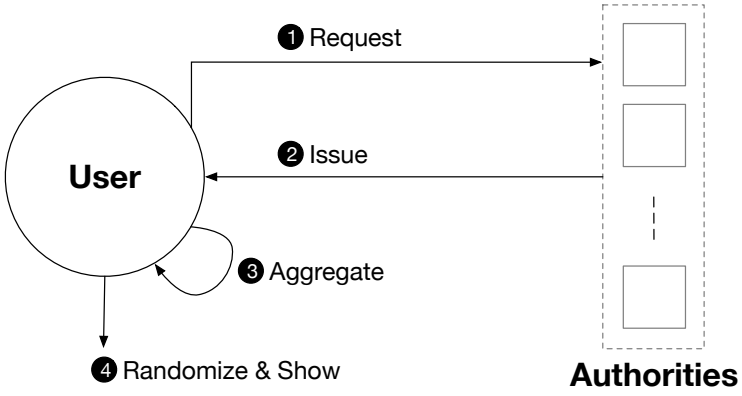


**Figure 6.2:** A high-level overview of Coconut architecture.

previously discussed applications; Section 6.7 presents a comparison with related work; and Section 6.9 concludes the chapter.

## 6.1 Overview

Coconut is a selective disclosure credential system, supporting threshold credential issuance of public and private attributes, re-randomization of credentials to support multiple unlinkable revelations, and the ability to selectively disclose a subset of attributes. It is embedded into a smart contract library that can be called from other contracts to issue credentials. The Coconut architecture is illustrated in Figure 6.2. Any Coconut user may send a Coconut *request* command to a set of Coconut signing authorities; this command specifies a set of public or encrypted private attributes to be certified into the credential (❶). Then, each authority answers with an *issue* command delivering a partial credential (❷). Any user can collect a threshold number of shares, aggregate them to form a single consolidated credential, and re-randomize it (❸). The use of the credential for authentication is however restricted to a user who knows the private attributes embedded in the credential—such as a private key. The user who owns the credentials can then execute the *show* protocol to selectively disclose attributes or statements about them (❹). The showing protocol is publicly verifiable, and may be publicly recorded. Coconut has the following design goals:

- **Threshold authorities:** Only a subset of the authorities is required to issue partial credentials in order to allow the users to generate a consolidated creden-

tial [117]. The communication complexity of the *request* and *issue* protocol is thus $O(t)$, where $t$ is the size of the subset of authorities. It is impossible to generate a consolidated credential from fewer than $t$ partial credentials.

- **Blind issuance & unlinkability:** The authorities issue the credential without learning any additional information about the private attributes embedded in the credential. Furthermore, it is impossible to link multiple showings of the credentials with each other, or the issuing transcript, even if all the authorities collude (see Section 6.2.2).

- **Non-interactivity:** The authorities may operate independently of each other, following a simple key distribution and setup phase to agree on public security and cryptographic parameters—they do not need to synchronize or further coordinate their activities.

- **Liveness:** Coconut guarantees liveness as long as a threshold number of authorities remains honest and weak synchrony assumptions holds for the key distribution [175].

- **Efficiency:** The credentials and all zero-knowledge proofs involved in the protocols are short and computationally efficient. After aggregation and re-randomization, the attribute showing and verification involve only a single consolidated credential, and are therefore $O(1)$ in terms of both cryptographic computations and communication of cryptographic material—no matter the number of authorities.

- **Short credentials:** Each partial credential—as well as the consolidated credential—is composed of exactly two group elements, no matter the number of authorities or the number of attributes embedded in the credentials.

As a result, a large number of authorities may be used to issue credentials, without significantly affecting efficiency.

## 6.2 The Coconut Construction

We introduce the cryptographic primitives supporting the Coconut architecture, step by step from the design of Pointcheval and Sanders [109] and Boneh *et al.* [113, 116] to the full Coconut scheme.

- **Step 1:** We first recall (Section 6.2.3) the scheme of Pointcheval *et al.* [109] for single-attribute credentials. We present its limitations preventing it from meeting our design goals presented in Section 6.1, and we show how to incorporate principles from Boneh *et al.* [113] to overcome them.
- **Step 2:** We introduce (Section 6.2.4) the *Coconut threshold credentials scheme*, which has all the properties of Pointcheval and Sanders [109] and Boneh *et al.* [113], and allows us to achieve all our design goals.
- **Step 3:** Finally, we extend (Section 6.2.5) our schemes to support credentials embedding $q$ distinct attributes $(m_1, \ldots, m_q)$ simultaneously.

### 6.2.1 Notations & Assumptions

We present the notation used in the rest of the paper, as well as the security assumptions on which our primitives rely.

**Zero-knowledge proofs**. Our credential scheme uses non-interactive zero-knowledge proofs to assert knowledge and relations over discrete logarithm values. We represent these non-interactive zero-knowledge proofs with the notation introduced by Camenisch *et al.* [176]:

$$\text{NIZK}\{(x, y, \ldots) : \text{statements about } x, y, \ldots\}$$

which denotes proving in zero-knowledge that the secret values $(x, y, \ldots)$ (all other values are public) satisfy the statements after the colon.

**Cryptographic assumptions**. Coconut requires groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order $p$ with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and satisfying (i) *Bilinearity*, (ii) *Non-degeneracy*, and (iii) *Efficiency* as described in Section 2.5. Coconut also relies on

a cryptographically secure hash function $H^*$, hashing an element $\mathbb{G}_1$ into an other element of $\mathbb{G}_1$, namely $H^* : \mathbb{G}_1 \to \mathbb{G}_1$. We implement this function by serializing the $(x, y)$ coordinates of the input point and applying a full-domain hash function to hash this string into an element of $\mathbb{G}_1$ (as Boneh *et al.* [113]).

**Threshold and communication assumptions**. Coconut assumes honest majority ($n/2 < t$) to prevent malicious authorities from issuing credentials arbitrarily. Coconut authorities do not need to communicate with each other; users wait for $t$-out-of-$n$ replies (in any order of arrival) and aggregate them into a consolidated credential; thus Coconut implicitly assumes an asynchronous setting. However, our current implementations rely on the distributed key generation protocol of Kate *et al.* [175], which requires (i) weak synchrony for liveness (but not for safety), and (ii) at most one third of dishonest authorities.

### 6.2.2 Scheme Definitions and Security Properties

We present the protocols that comprise a threshold credentials scheme:

❖ **Setup**$(1^\lambda) \to (params)$**:** defines the system parameters *params* with respect to the security parameter $\lambda$. These parameters are publicly available.

❖ **KeyGen**$(params) \to (sk, vk)$**:** is run by the authorities to generate their secret key *sk* and verification key *vk* from the public *params*.

❖ **AggKey**$(vk_1, \ldots, vk_t) \to (vk)$**:** is run by whoever wants to verify a credential to aggregate any subset of $t$ verification keys $vk_i$ into a single consolidated verification key *vk*. AggKey needs to be run only once.

❖ **IssueCred**$(m, \phi) \to (\sigma)$**:** is an interactive protocol between a user and each authority, by which the user obtains a credential $\sigma$ embedding the private attribute $m$ satisfying the statement $\phi$.

❖ **AggCred**$(\sigma_1, \ldots, \sigma_t) \to (\sigma)$**:** is run by the user to aggregate any subset of $t$ partial credentials $\sigma_i$ into a single consolidated credential.

❖ **ProveCred**$(vk, m, \phi') \rightarrow (\Theta, \phi')$**:** is run by the user to compute a proof $\Theta$ of possession of a credential certifying that the private attribute $m$ satisfies the statement $\phi'$ (under the corresponding verification key $vk$).

❖ **VerifyCred**$(vk, \Theta, \phi') \rightarrow (true/false)$**:** is run by whoever wants to verify a credential embedding a private attribute satisfying the statement $\phi'$, using the verification key $vk$ and cryptographic material $\Theta$ generated by ProveCred.

A threshold credential scheme must satisfy the following security properties:

**Unforgeability:** It must be unfeasible for an adversarial user to convince an honest verifier that they are in possession of a credential if they are in fact not (*i.e.*, if they have not received valid partial credentials from at least $t$ authorities).

**Blindness:** It must be unfeasible for an adversarial authority to learn any information about the attribute $m$ during the execution of the IssueCred protocol, except for the fact that $m$ satisfies $\phi$.

**Unlinkability / Zero-knowledge:** It must be unfeasible for an adversarial verifier (potentially working with an adversarial authority) to learn anything about the attribute $m$, except that it satisfies $\phi'$, or to link the execution of ProveCred with either another execution of ProveCred or with the execution of IssueCred (for a given attribute $m$).

### 6.2.3 Foundations of Coconut

Before giving the full Coconut construction, we first recall the credentials scheme proposed by Pointcheval and Sanders [109]; their construction has the same properties as CL-signatures [107] but is more efficient. The scheme works in a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of type 3, with a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ as described in Section 6.2.1.

❖ **P.Setup**$(1^\lambda) \rightarrow (params)$**:** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order $p$, where $p$ is a $\lambda$-bit prime number. Let $g_1$ be a generator of $\mathbb{G}_1$, and $g_2$ a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$.

❖ **P.KeyGen(*params*) → (*sk*, *vk*):** Choose a random secret key $sk = (x, y) \in \mathbb{F}_p^2$. Parse $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2)$, and publish the verification key $vk = (g_2, \alpha, \beta) = (g_2, g_2^x, g_2^y)$.

❖ **P.Sign(*params*, *sk*, *m*) → (σ):** Parse $sk = (x, y)$. Pick a random $r \in \mathbb{F}_p$ and set $h = g_1^r$. Output $\sigma = (h, s) = (h, h^{x+y \cdot m})$.

❖ **P.Verify(*params*, *vk*, *m*, σ) → (*true*/*false*):** Parse $vk = (g_2, \alpha, \beta)$ and $\sigma = (h, s)$. Output *true* if $h \neq 1$ and $e(h, \alpha\beta^m) = e(s, g_2)$; otherwise output *false*.

The signature $\sigma = (h, s)$ is randomizable by choosing a random $r' \in \mathbb{F}_p$ and computing $\sigma' = (h^{r'}, s^{r'})$. The above scheme can be modified to obtain credentials on a private attribute: to run IssueCred the user first picks a random $t \in \mathbb{F}_p$, computes the commitment $c_p = g_1^t Y^m$ to the message $m$, where $Y = g_1^y$; and sends it to a single authority along with a zero-knowledge proof of the opening of the commitment. The authority verifies the proof, picks a random $u \in \mathbb{F}_p$, and returns $\widetilde{\sigma} = (h, \widetilde{s}) = (g^u, (X c_p)^u)$ where $X = g_1^x$. The user unblinds the signature by computing $\sigma = (h, \widetilde{s}(h)^{-t})$, and this value acts as the credential.

This scheme provides blindness, unlinkability, efficiency and short credentials; but it does not support threshold issuance and therefore does not achieve our design goals. This limitation comes from the P.Sign algorithm—the issuing authority computes the credentials using a private and self-generated random number $r$ which prevents the scheme from being efficiently distributed to a multi-authority setting[1]. To overcome that limitation, we take advantage of a concept introduced by BLS signatures [113]; exploiting a hash function $H^* : \mathbb{F}_p \to \mathbb{G}_1$ to compute the group element $h = H^*(m)$. The next section describes how Coconut incorporates these concepts to achieve all our design goals.

### 6.2.4 The Coconut Threshold Credential Scheme

We introduce the *Coconut* threshold credential scheme, allowing users to obtain a partial credential $\sigma_i$ on a private or public attribute $m$. In a system with $n$ authorities,

[1]The original paper of Pointcheval and Sanders [109] proposes a sequential aggregate signature protocol that is unsuitable for threshold credentials issuance (see Section 6.7).

a $t$-out-of-$n$ threshold credentials scheme offers great flexibility as the users need to collect only $n/2 < t \leq n$ of these partial credentials in order to recompute the consolidated credential (both $t$ and $n$ are scheme parameters).

**Cryptographic primitives**. For the sake of simplicity, we describe below a key generation algorithm TTPKeyGen as executed by a trusted third party; this protocol can however be executed in a distributed way as illustrated by Gennaro *et al.* [177] under a synchrony assumption, and as illustrated by Kate *et al.* [175] under a weak synchrony assumption. Adding and removing authorities implies a re-run of the key generation algorithm—this limitation is inherited from the underlying Shamir's secret sharing protocol [178] and can be mitigated using techniques introduced by Herzberg *et al.* [179].

❖ **Setup**$(1^\lambda) \rightarrow$ (*params*)**:** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order $p$, where $p$ is a $\lambda$-bit prime number. Let $g_1, h_1$ be generators of $\mathbb{G}_1$, and $g_2$ a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h_1)$.

❖ **TTPKeyGen**$(params, t, n) \rightarrow (sk, vk)$**:** Pick[2] two polynomials $v, w$ of degree $t-1$ with coefficients in $\mathbb{F}_p$, and set $(x, y) = (v(0), w(0))$. Issue to each authority $i \in [1, \ldots, n]$ a secret key $sk_i = (x_i, y_i) = (v(i), w(i))$, and publish their verification key $vk_i = (g_2, \alpha_i, \beta_i) = (g_2, g_2^{x_i}, g_2^{y_i})$.

❖ **IssueCred**$(m, \phi) \rightarrow (\sigma)$**:** Credentials issuance is composed of three algorithms:

❖ PrepareBlindSign$(m, \phi) \rightarrow (d, \Lambda, \phi)$: The users generate an El-Gamal keypair $(d, \gamma = g_1^d)$; pick a random $o \in \mathbb{F}_p$, compute the commitment $c_m$ and the group element $h \in \mathbb{G}_1$ as follows:

$$c_m = g_1^m h_1^o \qquad \text{and} \qquad h = H^*(c_m)$$

Pick a random $k \in \mathbb{F}_p$ and compute an El-Gamal encryption of $m$ as below:

$$c = Enc(h^m) = (g_1^k, \gamma^k h^m)$$

[2]This algorithm can be turned into the KeyGen and AggKey algorithms described in Section 6.2.2 using techniques illustrated by Gennaro *et al.* [177] or Kate *et al.* [175].

Output $(d, \Lambda = (\gamma, c_m, c, \pi_s), \phi)$, where $\phi$ is an application-specific predicate satisfied by $m$, and $\pi_s$ is defined by:

$$\pi_s = \text{NIZK}\{(d, m, o, k) : \gamma = g_1^d \ \wedge \ c_m = g_1^m h_1^o \\ \wedge \ c = (g_1^k, \gamma^k h^m) \ \wedge \ \phi(m) = 1\}$$

- ❖ $\mathsf{BlindSign}(sk_i, \Lambda, \phi) \rightarrow (\tilde{\sigma}_i)$: The authority $i$ parses $\Lambda = (\gamma, c_m, c, \pi_s)$, $sk_i = (x_i, y_i)$, and $c = (a, b)$. Recompute $h = H^*(c_m)$. Verify the proof $\pi_s$ using $\gamma$, $c_m$ and $\phi$; if the proof is valid, build $\tilde{c}_i = (a^y, h^{x_i} b^{y_i})$ and output $\tilde{\sigma}_i = (h, \tilde{c}_i)$; otherwise output $\perp$ and stop the protocol.
- ❖ $\mathsf{Unblind}(\tilde{\sigma}_i, d) \rightarrow (\sigma_i)$: The users parse $\tilde{\sigma}_i = (h, \tilde{c})$ and $\tilde{c} = (\tilde{a}, \tilde{b})$; compute $\sigma_i = (h, \tilde{b}(\tilde{a})^{-d})$. Output $\sigma_i$.

❖ **$\mathsf{AggCred}(\sigma_1, \ldots, \sigma_t) \rightarrow (\sigma)$:** Parse each $\sigma_i$ as $(h, s_i)$ for $i \in [1, \ldots, t]$. Output $(h, \prod_{i=1}^{t} s_i^{l_i})$, where $l$ is the Lagrange coefficient:

$$l_i = \left[ \prod_{j=1, j \neq i}^{t} (0 - j) \right] \left[ \prod_{j=1, j \neq i}^{t} (i - j) \right]^{-1} \mod p$$

❖ **$\mathsf{ProveCred}(vk, m, \sigma, \phi') \rightarrow (\Theta, \phi')$:** Parse $\sigma = (h, s)$ and $vk = (g_2, \alpha, \beta)$. Pick at random $r', r \in \mathbb{F}_p^2$; set $\sigma' = (h', s') = (h^{r'}, s^{r'})$; build $\kappa = \alpha \beta^m g_2^r$ and $\nu = (h')^r$. Output $(\Theta = (\kappa, \nu, \sigma', \pi_v), \phi')$, where $\phi'$ is an application-specific predicate satisfied by $m$, and $\pi_v$ is:

$$\pi_v = \text{NIZK}\{(m, r) : \kappa = \alpha \beta^m g_2^r \ \wedge \ \nu = \left(h'\right)^r \ \wedge \ \phi'(m) = 1\}$$

❖ **$\mathsf{VerifyCred}(vk, \Theta, \phi') \rightarrow (true/false)$:** Parse $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and $\sigma' = (h', s')$; verify $\pi_v$ using $vk$ and $\phi'$. Output $true$ if the proof verifies, $h' \neq 1$ and $e(h', \kappa) = e(s' \nu, g_2)$; otherwise output $false$.

**Correctness and explanation**. The Setup algorithm generates the public parameters. Credentials are elements of $\mathbb{G}_1$, while verification keys are elements of $\mathbb{G}_2$. Figure 6.3

illustrates the protocol exchanges. To keep an attribute $m \in \mathbb{F}_p$ hidden from the authorities, the users run PrepareBlindSign to produce $\Lambda = (\gamma, c_m, c, \pi_s)$. They create an El-Gamal keypair $(d, \gamma = g_1^d)$, pick a random $o \in \mathbb{F}_p$, and compute a commitment $c_m = g_1^m h_1^o$. Then, the users compute $h = H^*(c_m)$ and the encryption of $h^m$ as below:

$$c = Enc(h^m) = (a, b) = (g_1^k, \gamma^k h^m),$$

where $k \in \mathbb{F}_p$. Finally, the users send $(\Lambda, \phi)$ to the signer, where $\pi_s$ is a zero-knowledge proof ensuring that $m$ satisfies the application-specific predicate $\phi$, and correctness of $\gamma, c_m, c$ (❶). All the zero-knowledge proofs required by Coconut are based on standard sigma protocols to show knowledge of representation of discrete logarithms; they are based on the DH assumption [176] and do not require any trusted setup. To blindly sign the attribute, each authority $i$ verifies the proof $\pi_s$, and uses the homomorphic properties of El-Gamal to generate an encryption $\tilde{c}$ of $h^{x_i + y_i \cdot m}$ as below:

$$\tilde{c} = (a^y, h^{x_i} b^{y_i}) = (g_1^{k y_i}, \gamma^{k y_i} h^{x_i + y_i \cdot m})$$

Note that every authority must operate on the same element $h$. Intuitively, generating $h$ from $h = H^*(c_m)$ is equivalent to computing $h = g_1^{\tilde{r}}$ where $\tilde{r} \in \mathbb{F}_p$ is unknown by the users (as in Pointcheval and Sanders [109]). However, since $h$ is deterministic, every authority can uniquely derive it in isolation and forgeries are prevented since different $m_0$ and $m_1$ cannot lead to the same value of $h$.[3] As described in Section 6.2.3, the blind signature scheme of Pointcheval and Sanders builds the credentials directly from a commitment of the attribute and a blinding factor secretly chosen by the authority; this is unsuitable for issuance of threshold credentials. We circumvent that problem by introducing the El-Gamal ciphertext $c$ in our scheme and exploiting its homomorphism, as described above.

Upon reception of $\tilde{c}$, the users decrypt it using their El-Gamal private key $d$ to recover the partial credentials $\sigma_i = (h, h^{x_i + y_i \cdot m})$; this is performed by the Unblind

[3] If an adversary $\mathcal{A}$ can obtain two credentials $\sigma_0$ and $\sigma_1$ on respectively $m_0 = 0$ and $m_1 = 1$ with the same value $h$ as follows: $\sigma_0 = h^x$ and $\sigma_1 = h^{x+y}$; then $\mathcal{A}$ could forge a new credential $\sigma_2$ on $m_2 = 2$: $\sigma_2 = (\sigma_0)^{-1} \sigma_1 \sigma_1 = h^{x+2y}$.

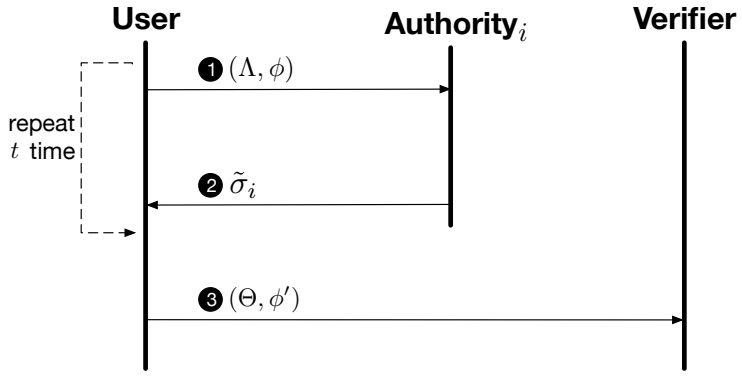


**Figure 6.3:** Coconut threshold credentials protocol exchanges.

algorithm (❷). Then, the users can call the AggCred algorithm to aggregate any subset of $t$ partial credentials. This algorithm uses the Lagrange basis polynomial $l$ which allows to reconstruct the original $v(0)$ and $w(0)$ through polynomial interpolation;

$$v(0) = \sum_{i=1}^{t} v(i) l_i \quad \text{and} \quad w(0) = \sum_{i=1}^{t} w(i) l_i$$

However, this computation happens in the exponent—neither the authorities nor the users should know the values $v(0)$ and $w(0)$. One can easily verify the correctness of AggCred of $t$ partial credentials $\sigma_i = (h_i, s_i)$ as below.

$$\begin{aligned} s &= \prod_{i=1}^{t} (s_i)^{l_i} = \prod_{i=1}^{t} \left(h^{x_i + y_i \cdot m}\right)^{l_i} \\ &= \prod_{i=1}^{t} (h^{x_i})^{l_i} \prod_{i=1}^{t} (h^{y_i \cdot m})^{l_i} = \prod_{i=1}^{t} h^{(x_i l_i)} \prod_{i=1}^{t} h^{(y_i l_i) \cdot m} \\ &= h^{v(0) + w(0) \cdot m} = h^{x + y \cdot m} \end{aligned}$$

Before verification, the verifier collects and aggregates the verifications keys of the authorities—this process happens only once and ahead of time. The algorithms ProveCred and VerifyCred implement verification. First, the users randomize the credentials by picking a random $r' \in \mathbb{F}_p$ and computing $\sigma' = (h', s') = (h^{r'}, s^{r'})$; then, they compute $\kappa$ and $\nu$ from the attribute $m$, a blinding factor $r \in \mathbb{F}_p$ and the

aggregated verification key:

$$\kappa = \alpha\beta^m g_2^r \qquad \text{and} \qquad \nu = (h')^r$$

Finally, they send $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and $\phi'$ to the verifier where $\pi_v$ is a zero-knowledge proof asserting the correctness of $\kappa$ and $\nu$; and that the private attribute $m$ embedded into $\sigma$ satisfies the application-specific predicate $\phi'$ (❸). The proof $\pi_v$ also ensures that the users actually know $m$ and that $\kappa$ has been built using the correct verification keys and blinding factors. The pairing verification is similar to Pointcheval and Sanders [109] and Boneh *et al.* [113]; expressing $h' = g_1^{\tilde{r}} \mid \tilde{r} \in \mathbb{F}_p$, the left-hand side of the pairing verification can be expanded as:

$$e(h', \kappa) = e(h', g_2^{(x+my+r)}) = e(g_1, g_2)^{(x+my+r)\tilde{r}}$$

and the right-hand side:

$$e(s'\nu, g_2) = e(h'^{(x+my+r)}, g_2) = e(g_1, g_2)^{(x+my+r)\tilde{r}}$$

From where the correctness of VerifyCred follows.

**Security**. The proof system we require is based on standard sigma protocols to show knowledge of representation of discrete logarithms, and can be rendered non-interactive using the Fiat-Shamir heuristic [111] in the random oracle model. As our signature scheme is derived from the ones due to Pointcheval and Sanders [109] and BLS [113], we inherit their assumptions; namely, LRSW [180] and XDH [113].

**Security Theorem 10.** *Assuming LRSW, XDH, and the existence of random oracles, Coconut is a secure threshold credentials scheme, meaning it satisfies unforgeability (as long as fewer than t authorities collude), blindness, and unlinkability.*

A sketch of this proof, based on the security of the underlying components of Coconut, can be found in Section 6.3. Coconut guarantees unforgeability as long as less than $t$ authorities collude ($t > n/2$), and guarantees blindness and unlinkability no matter how many authorities collude (and even if the verifier colludes with the

authorities).

### 6.2.5 Multi-Attribute Credentials

We expand our scheme to embed multiple attributes into a single credential without increasing its size; this generalization follows directly from the Waters signature scheme [114] and Pointcheval and Sanders [109]. The authorities' key pairs become:

$$sk = (x, y_1, \ldots, y_q) \quad \text{and} \quad vk = (g_2, g_2^x, g_2^{y_1}, \ldots, g_2^{y_q})$$

where $q$ is the number of attributes. The multi-attribute credential is derived from the commitment $c_m$ and the group element $h$ as below:

$$c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \qquad \text{and} \qquad h = H^*(c_m)$$

and the credential generalizes as follows:

$$\sigma = (h, h^{x+\Sigma_{j=1}^{q} m_j y_j})$$

The credential's size does not increase with the number of attributes or authorities—it is always composed of two group elements. The security proof of the multi-attribute scheme relies on a reduction against the single-attribute scheme and is analogous to Pointcheval and Sanders [109]. Moreover, it is also possible to combine public and private attributes to keep only a subset of the attributes hidden from the authorities, while revealing some others; the BlindSign algorithm only verifies the proof $\pi_s$ on the private attributes (similar to Chase *et al.* [108]). If the credentials include only non-random attributes, the verifier could guess its value by brute-forcing the verification algorithm[4]. This issue is prevented by always embedding a private random attribute into the credentials, that can also act as the authorization key for the credential.

**Cryptographic primitives**. As in Section 6.2.4, we describe below a key generation

[4]Let assume for example that some credentials include a single attribute $m$ representing the age of the user; the verifier can run the verification algorithm $e(h, \kappa(\alpha \cdot \beta^m)^{-1}) = e(\nu, g_2)$ for every $m \in [1, 100]$ and guess the value of $m$.

algorithm TTPKeyGen as executed by a trusted third party; this protocol can however be execute in a distributed way as illustrated by Kate *et al.* [175].

❖ **Setup**$(1^\lambda, q) \rightarrow$ (*params*)**:** Choose a bilinear group $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ with order $p$, where $p$ is an $\lambda$-bit prime number. Let $g_1, h_1, \ldots, h_q$ be generators of $\mathbb{G}_1$, and $g_2$ a generator of $\mathbb{G}_2$. The system parameters are $params = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, h_1, \ldots, h_q)$.

❖ **TTPKeyGen**$(params, t, n, q) \rightarrow (sk, vk)$**:** Choose $(q+1)$ polynomials of degree $(t-1)$ with coefficients in $\mathbb{F}_p$, noted $(v, w_1, \ldots, w_q)$, and set:

$$(x, y_1, \ldots y_q) = (v(0), w_1(0), \ldots, w_q(0))$$

Issue a secret key $sk_i$ to each authority $i \in [1, \ldots, n]$ as below:

$$sk_i = (x_i, y_{i,1}, \ldots, y_{i,q}) = (v(i), w_1(i), \ldots, w_q(i))$$

and publish their verification key $vk_i$ computed as follows:

$$vk_i = (g_2, \alpha_i, \beta_{i,1}, \ldots, \beta_{i,q)}) = (g_2, g_2^{x_i}, g_2^{y_{i,1}}, \ldots, g_2^{y_{i,1q}})$$

❖ **IssueCred**$(m_1, \ldots, m_q, \phi) \rightarrow (\sigma)$**:** Credentials issuance is composed of three algorithms:

❖ PrepareBlindSign$(m_1, \ldots, m_q, \phi) \rightarrow (d, \Lambda, \phi)$: The users generate an El-Gamal key-pair $(d, \gamma = g_1^d)$; pick a random $o \in \mathbb{F}_p$ compute the commitment $c_m$ and the group element $h \in \mathbb{G}_1$ as follows:

$$c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \qquad \text{and} \qquad h = H^*(c_m)$$

Pick at random $(k_1, \ldots, k_q) \in \mathbb{F}_p^q$ and compute an El-Gamal encryption of each $m_j$ for $\forall j \in [1, \ldots, q]$ as below:

$$c_j = Enc(h^{m_j}) = (g_1^{k_j}, \gamma^{k_j} h^{m_j})$$

Output $(d, \Lambda = (\gamma, c_m, c_j, \pi_s), \phi)$ $\forall j \in [1, \ldots, q]$, where $\pi_s$ is defined by:

$$\begin{aligned}\pi_s = &\text{NIZK}\{(d, m_1, \ldots, m_q, o, k_1, \ldots, k_q) : \gamma = g_1^d \\ &\wedge\ c_m = g_1^o \prod_{j=1}^{q} h_j^{m_j} \ \wedge\ c_j = (g_1^{k_j}, \gamma^{k_j} h^{m_j}) \\ &\wedge\ \phi(m_1, \ldots, m_q) = 1\} \quad \forall j \in [1, \ldots, q]\end{aligned}$$

❖ $\mathsf{BlindSign}(sk, \Lambda, \phi) \rightarrow (\tilde{\sigma}_i)$: The authority $i$ parses $\Lambda = (\gamma, c_m, c_j, \pi_s)$ and $c_j = (a_j, b_j)$ $\forall j \in [1, \ldots, q]$, and $sk_i = (x, y_1, \ldots, y_q)$. Recompute $h = H^*(c_m)$. Verify the proof $\pi_s$ using $\gamma, c_m$ and $\phi$. If the proof is invalid, output $\perp$ and stop the protocol; otherwise output $\tilde{\sigma}_i = (h, \tilde{c})$, where $\tilde{c}$ is defined as below:

$$\tilde{c} = \left( \prod_{j=1}^{q} a_j^{y_j}, h^x \prod_{j=1}^{q} b_j^{y_j} \right)$$

❖ $\mathsf{Unblind}(\tilde{\sigma}_i, d) \rightarrow (\sigma_i)$: The users parse $\tilde{\sigma}_i = (h, \tilde{c})$ and $\tilde{c} = (\tilde{a}, \tilde{b})$; compute $\sigma_i = (h, \tilde{b}(\tilde{a})^{-d})$. Output $\sigma_i$.

❖ **$\mathsf{AggCred}(\sigma_1, \ldots, \sigma_t) \rightarrow (\sigma)$:** Parse each $\sigma_i$ as $(h, s_i)$ for $i \in [1, \ldots, t]$. Output $(h, \prod_{i=1}^{t} s_i^{l_i})$, where:

$$l_i = \left[ \prod_{i=1, j \neq i}^{t} (0 - j) \right] \left[ \prod_{i=1, sj \neq i}^{t} (i - j) \right]^{-1} \bmod p$$

❖ **$\mathsf{ProveCred}(vk, m_1, \ldots, m_q, \sigma, \phi') \rightarrow (\sigma', \Theta, \phi')$:** Parse $\sigma = (h, s)$ and $vk = (g_2, \alpha, \beta_1, \ldots, \beta_q)$. Pick at random $r', r \in \mathbb{F}_q^2$; set $\sigma' = (h', s') = (h^{r'}, s^{r'})$, and build $\kappa$ and $\nu$ as below:

$$\kappa = \alpha \prod_{j=1}^{q} \beta_j^{m_j} g_2^r \qquad \text{and} \qquad \nu = \left(h'\right)^r$$

Output $(\Theta = (\kappa, \nu, \sigma', \pi_v), \phi')$, where $\pi_v$ is:

$$\pi_v = \text{NIZK}\{(m_1, \ldots, m_q, r) : \kappa = \alpha \prod_{j=1}^{q} \beta_j^{m_j} g_2^r$$
$$\wedge\ \nu = \left(h'\right)^r \wedge\ \phi(m_1, \ldots, m_q) = 1\}$$

❖ **VerifyCred**$(vk, \Theta, \phi') \rightarrow (true/false)$**:** Parse $\Theta = (\kappa, \nu, \sigma', \pi_v)$ and $\sigma' = (h', s')$; verify $\pi_v$ using $vk$ and $\phi'$; Output $true$ if the proof verifies, $h' \neq 1$ and $e(h', \kappa) = e(s'\nu, g_2)$; otherwise output $false$.

## 6.3 Sketch of Security Proofs

This section sketches the security proofs of the cryptographic construction described in Section 6.2.

**Unforgeability**. There are two possible ways for an adversary to forge a proof of a credential: (i) an adversary without a valid credential nevertheless manages to form a proof such that VerifyCred passes; and (ii), an adversary that has successfully interacted with fewer than $t$ authorities generates a valid consolidated credential (of which they then honestly prove possession using ProveCred).

Unforgeability in scenario (i) is ensured by the soundness property of the zero-knowledge proof. For scenario (ii), running AggCred involves performing Lagrange interpolation. If an adversary has fewer than $t$ partial credentials, then they have fewer than $t$ points, which makes the resulting polynomial (of degree $t-1$) undetermined and information-theoretically impossible to compute. The only option available to the adversary is thus to forge the remaining credentials directly. This violates the unforgeability of the underlying blind signature scheme, which was proved secure by Pointcheval and Sanders [109] under the LRSW assumption [180].

**Blindness**. Blindness follows directly from the blindness of the signature scheme used during IssueCred, which was largely proved secure by Pointcheval and Sanders [109] under the XDH assumption [113]. There are only two differences between their protocol and ours.

First, the Coconut authorities generate the credentials from a group element $h = H^*(c_m)$ instead of from $g_1^{\tilde{r}}$ for random $\tilde{r} \in \mathbb{F}_p$. The hiding property of the commitment $c_m$, however, ensures that $H^*(c_m)$ does not reveal any information about $m$. Second, Pointcheval and Sanders use a commitment to the attributes as input to BlindSign (see Section 6.2.3), whereas Coconut uses an encryption instead. The IND-CPA property, however, of the encryption scheme ensures that the ciphertext also reveals no information about $m$. Concretely, Coconut uses Pedersen Commitments [131] for the commitment scheme, which is secure under the discret logarithm assumption. It uses El-Gamal for the encryption scheme in $\mathbb{G}_1$, which is secure assuming DDH. Finally, it relies on the blindness of the Pointcheval and Sanders signature, which is secure assuming XDH [113]. As XDH implies both of the previous two assumptions, our entire blindness argument is implied by XDH.

**Unlinkability / Zero-knowledge**. Unlinkability and zero-knowledge are guaranteed under the XDH assumption [113]. The zero-knowledge property of the underlying proof system ensures that ProveCred does not on its own reveal anything more than the validity of the statement $\phi'$, which may include public attributes (see Section 6.2.5. The fact that credentials are re-randomized at the start of ProveCred in turn ensures unlinkability, both between different executions of ProveCred and between an execution of ProveCred and of IssueCred.

## 6.4 Implementation

We implement a Python library for Coconut as described in Section 6.2 and publish the code on GitHub as an open-source project[5]. We also implement a smart contract library in Chainspace [25] to enable other application-specific smart contracts (see Section 6.5) to conveniently use our cryptographic primitives. We present the design and implementation of the Coconut smart contract library in Section 6.4.1. In addition, we implement and evaluate some of the functionality of the Coconut smart contract library in Ethereum [24] (Section 6.4.2). Finally, we show how to integrate Coconut into existing semi-permissioned blockchains (Section 6.4.3).

[5] `https://github.com/asonnino/coconut`

### 6.4.1 The Coconut Smart Contract Library

We implement the Coconut smart contract in Chainspace[6] (which can be used by other application-specific smart contracts) as a library to issue and verify randomizable threshold credentials through cross-contract calls. Running this library as an independent smart contract enables other application-specific smart contracts to securely delegate the credentials issuance and verification processes to the library through a cross-contract call. The contract has four functions, (Create, Request, Issue, Verify), as illustrated in Figure 6.4. First, a set of authorities call the Create function to initialize a Coconut instance defining the *contract info*; *i.e.* their verification key, the number of authorities and the threshold parameter (❶). The initiator smart contract can specify a callback contract that needs to be executed by the user in order to request credentials; *e.g.* this callback can be used for authentication. The instance is public and can be read by the user (❷); any user can request a credential through the Request function by executing the specified callback contract, and providing the public and private *attributes* to include in the credentials (❸). The public attributes are simply a list of clear text strings, while the private attributes are encrypted as described in Section 6.2.4. Each signing authority monitors the blockchain at all times, looking for credential requests. If the request appears on the blockchain (*i.e.* a transaction is executed), it means that the callback has been correctly executed (❹); each authority issues a partial *credential* on the specified attributes by calling the Issue procedure (❺). In our implementation, all partial credentials are in the blockchain; however, these can also be provided to the user off-chain. Users collect a threshold number of partial credentials, and aggregate them to form a full credential (❻). Then, the users locally randomize the credential. The last function of the Coconut library contract is Verify that allows the blockchain—and anyone else—to check the validity of a given credential (❼).

A limitation of this architecture is that it is not efficient for the authorities to continuously monitor the blockchain. Section 6.4.3 explains how to overcome this limitation by embedding the authorities into the nodes running the blockchain.

[6] `https://github.com/asonnino/coconut-chainspace`

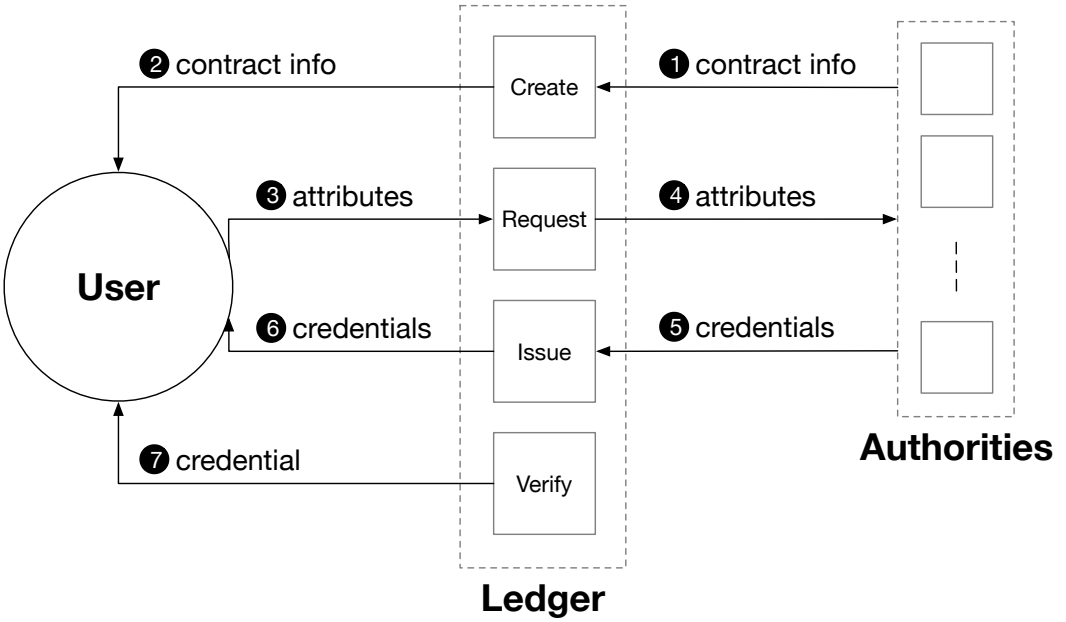


**Figure 6.4:** The Coconut smart contract library.

### 6.4.2 Ethereum Smart Contract Library

To make Coconut more widely available, we also implement it in Ethereum—a popular permissionless smart contract blockchain [24]. We release the Coconut Ethereum smart contract as an open source library[7]. The library is written in Solidity, a high-level JavaScript-like language that compiles down to Ethereum Virtual Machine (EVM) assembly code. Ethereum recently hardcoded a pre-compiled smart contract in the EVM for performing pairing checks and elliptic curve operations on the alt_bn128 curve [181, 182], for efficient verification of zkSNARKs. The execution of an Ethereum smart contract has an associated 'gas cost', a fee that is paid to miners for executing a transaction. Gas cost is calculated based on the operations executed by the contract; *i.e.* the more operations, the higher the gas cost. The pre-compiled contracts have lower gas costs than equivalent native Ethereum smart contracts. We use the pre-compiled contract for performing a pairing check, in order to implement Coconut verification within a smart contract. The Ethereum code only implements elliptic curve addition and scalar multiplication on $\mathbb{G}_1$, whereas Coconut requires operations on $\mathbb{G}_2$ to verify credentials. Therefore, we implement elliptic curve addition and scalar multiplication on $\mathbb{G}_2$ as an Ethereum smart contract

[7] `https://github.com/musalbas/coconut-ethereum`

library written in Solidity that we also release open source[8]. This is a practical solution for many Coconut applications, as verifying credentials with one revealed attribute only requires one addition and one scalar multiplication. It would not be practical however to verify credentials with attributes that will not be revealed—this requires three $\mathbb{G}_2$ multiplications using our elliptic curve implementation, which would exceed the current Ethereum block gas limit (10M as of September 2019).

We can however use the Ethereum contract to design a federated peg for side chains, or a coin tumbler as an Ethereum smart contract, based on credentials that reveal one attribute. We go on to describe and implement this tumbler using the Coconut Chainspace library in Section 6.5.1, however the design for the Ethereum version differs slightly to avoid the use of attributes that will not be revealed. The library shares the same functions as the Chainspace library described in Section 6.4.1, except for Request and Issue which are computed off the blockchain to save gas costs. As Request and Issue functions simply act as a communication channel between users and authorities, users can directly communicate with authorities off the blockchain to request tokens. This saves significant gas costs that would be incurred by storing these functions on the blockchain. The Verify function simply verifies tokens against Coconut instances created by the Create function.

### 6.4.3 Deeper Blockchain Integration

The designs described in Section 6.4.1 and Section 6.4.2 rely on authorities on-the-side for issuing credentials. In this section, we present designs that incorporate Coconut authorities within the infrastructure of a number of semi-permissioned blockchains. This enables the issuance of credentials as a side effect of the normal system operations, taking no additional dependency on extra authorities. It remains an open problem how to embed Coconut into permissionless systems. These systems have a highly dynamic set of nodes maintaining the state of their blockchains, which cannot readily be mapped into Coconut issuing authorities.

Integration of Coconut into Hyperledger Fabric [39]—a permissioned blockchain platform—is straightforward. Fabric contracts run on private sets

[8] https://github.com/musalbas/solidity-BN256G2

of computation nodes—and use the Fabric protocols for cross-contract calls. In this setting, Coconut issuing authorities can coincide with the Fabric smart contract authorities. Upon a contract setup, they perform the setup and key distribution, and then issue partial credentials when authorized by the contract. For issuing Coconut credentials, the only secrets maintained are the private issuing keys; all other operations of the contract can be logged and publicly verified. Coconut has obvious advantages over using traditional CL credentials relying on a single authority—as currently present in the Hyperledger roadmap[9]. The threshold trust assumption—namely that integrity and availability is guaranteed under the corruption of a subset of authorities is preserved, and prevents forgeries by a single corrupted node. We can also naturally embed Coconut into sharded scalable blockchains, as exemplified by Chainspace [25] (which supports general smart contracts), and Omniledger [4] (which supports digital tokens). In these systems, transactions are distributed and executed on 'shards' of authorities, whose membership and public keys are known. Coconut authorities can naturally coincide with the nodes within a shard—a special transaction type in Omniledger, or a special object in Chainspace, can signal to them that issuing a credential is authorized. The authorities, then issue the partial signature necessary to reconstruct the Coconut credential, and attach it to the transaction they are processing anyway. Users can aggregate, re-randomize and show the credential.

## 6.5 Applications

In this section, we present three applications that leverage Coconut to offer improved security and privacy properties—a coin tumbler (Section 6.5.1), a privacy-preserving petition system (Section 6.5.2), and a system for censorship-resistant distribution of proxies (Section 6.5.3). For generality, the applications assume authorities external to the blockchain, but these can also be embedded into the blockchain as described in Section 6.4.3.

[9] `http://nick-fabric.readthedocs.io/en/latest/idemix.html`

### 6.5.1 Coin Tumbler

We implement a coin tumbler (or mixer) on Chainspace as depicted in Figure 6.5. Coin tumbling is a method to mix cryptocurrency associated with an address visible in a public ledger with other addresses, to "clean" the coins and obscure the trail back to the coins' original source address. A limitation of previous similar schemes [183, 184, 185, 186, 187, 188, 189] is that they are either centralized (*i.e.* there is a central authority that operates the tumbler, which may go offline), or require users to coordinate with each other. The Coconut tumbler addresses these issues *via* a distributed design (*i.e.* security relies on a set of multiple authorities that are collectively trusted to contain at least $t$ honest ones), and does not require users to coordinate with each other. Zcash [43] achieves a similar goal: it theoretically hides the totality of the transaction but at a large computational cost, and offers the option to cheaply send transactions in clear. In practice, the computational overhead of sending hidden transactions makes it impractical, and only a few users take advantage of the optional privacy provided by Zcash; as a result, transactions are easy to de-anonymize [190], and recent works aim to reduce the computational overhead of Zcash hidden transactions [191]. Coconut provides efficient proofs taking only a few milliseconds (see Section 6.6), and makes hidden transactions practical. Trust assumptions in Zcash are different from Coconut. However, instead of assuming a threshold number of honest authorities, Zcash relies on zk-SNARKs which assumes a setup algorithm executed by a trusted authority[10]. Möbius [189]—which was developed concurrently—is a coin tumbler based on Ethereum smart contracts that achieves strong notions of anonymity and low off-chain communication complexity. Möbius relies on ring signatures to allow parties to prove group membership without revealing exactly which public key belongs to them.

**Chainspace coin tumbler**. Our tumbler uses Coconut to instantiate a pegged side-chain [174], providing stronger value transfer anonymity than the original cryptocurrency platform, through unlinkability between issuing a credential representing an e-coin [193], and spending it. The tumbler application is based on the Coconut

[10]Recent proposals aim to distribute this trusted setup [192].

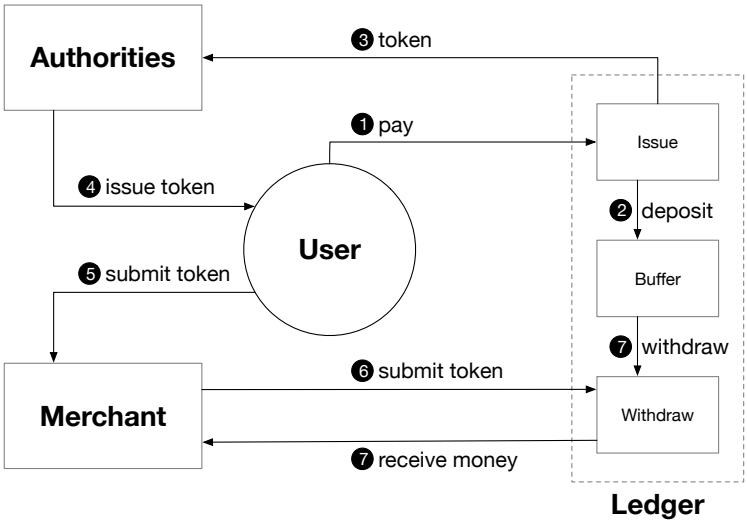


**Figure 6.5:** The coin tumbler application.

contract library and an application specific smart contract called 'tumbler'. A set of authorities jointly create an instance of the Coconut smart contract as described in Section 6.4.1 and specify the smart contract handling the coins of the underlying blockchain as callback. Specifically, the callback requires a coin transfer to a buffer account. Then users execute the callback and *pay* $v$ coins to the buffer to ask a credential on the public attribute $v$, and on two private attributes: the user's private key $k$ and a randomly generated sequence number $s$ (❶). Note that to prevent tracing traffic analysis, $v$ should be limited to a specific set of possible values (similar to cash denominations). The request is accepted by the blockchain only if the user *deposited* $v$ coins to the buffer account (❷). Each authority monitors the blockchain and detects the *request* (❸); and issues a partial *credential* to the user (either on chain or off-chain) (❹). The user aggregates all partial credentials into a consolidated credential, re-randomizes it, and *submits* it as coin token to a merchant. First, the user produces a zk-proof of knowledge of its private key by binding the proof to the merchant's address $addr$; then, the user provides the merchant with the proof along with the sequence number $s$ and the consolidated credential (❺). The coins can only be spent with knowledge of the associated sequence number and by the owner of $addr$. To accept the above as payment, the merchant *submits* the token by showing the credential and a group element $\zeta = g_1^s \in \mathbb{G}_1$ to the tumbler contract

along with a zero-knowledge proof ensuring that $\zeta$ is well-formed (❻). To prevent double spending, the tumbler contract keeps a record of all elements $\zeta$ that have already been shown. Upon showing a $\zeta$ embedding a fresh (unspent) sequence number $s$, the contract verifies that the credential and zero-knowledge proofs check, and that $\zeta$ doesn't already appear in the spent list. Then it *withdraws* $v$ coins from the buffer (❼), sends them to be *received* by the merchant account determined by $addr$, and adds $\zeta$ to the spent list (❽). For the sake of simplicity, we keep the transfer value $v$ in clear-text (treated as a public attribute), but this could be easily hidden by integrating a range proof; this can be efficiently implemented using the technique developed by Bünz *et al.* [194].

**Security consideration**. Coconut provides blind issuance which allows the user to obtain a credential on the sequence number $s$ without the authorities learning its value. Without blindness, any authority seeing the user key $k$ could potentially race the user and the merchant, and spend it—blindness prevents authorities from stealing the token. Furthermore, Coconut provides unlinkability between the *pay* phase (❶) and the *submit* phase (❺) (see Figure 6.5), and prevents any authority or third parties from keeping track of the user's transactions. As a result, a merchant can receive payments for good or services offered, yet not identify the purchasers. Keeping a spent list of all elements $\zeta$ prevents double-spending attacks [195] without revealing the sequence number $s$; this prevents an attacker from exploiting a race condition in the *submit token* phase (❻) and lock user's funds[11]. Finally, this application prevents a single authority from creating coins to steal all the money in the buffer. The threshold property of Coconut implies that the adversary needs to corrupt at least $t$ authorities for this attack to be possible. A small subset of authorities cannot block the issuance of a token—the service is guaranteed to be available as long as at least $t$ authorities are running.

**Adapting the coin tumbler to Ethereum**. We extend the example of the tumbler application described above to the Ethereum version of the Coconut library, with a

[11] An attacker observing a sequence number $s$ during a *submit token* phase (❻) could exploit a race condition to lock users fund by quickly buying a token using the same $s$, and spending it before the original *submit token* phase is over.

few modifications to reduce the gas costs. Instead of having $v$ (the number of coins) as an attribute, which would increase the number of elliptic curve multiplications required to verify the credentials, we allow for a fixed number of instances of Coconut to be setup for different denominations for $v$. The Tumbler has a Deposit method, where users deposit Ether into the contract, and then send an issuance request to authorities on one private attribute: $addr||s$, where $addr$ is the destination address of the merchant, and $s$ is a randomly generated sequence number (1). It is necessary for $addr$ to be a part of the attribute because once the attribute is revealed, the credential can be spent by anyone with knowledge of the attribute (including any peers monitoring the blockchain for transactions), therefore the credential must be bounded to a specific recipient address before it is revealed. This issuance request is signed by the Ethereum address that deposited the Ether into the smart contract, as proof that the request is associated with a valid deposit, and sent to the authorities (2). As $addr$ and $s$ will be both revealed at the same time when withdrawing the token, we concatenate these in one attribute to save on elliptic curve operations. Users aggregate the credentials issued by the authorities (3). The resulting token can then be passed to the Withdraw function, where the withdrawer reveals $addr$ and $s$ (4). As usual, the contract maintains a map of $s$ values associated with tokens that have already been withdrawn to prevent double-spending. After checking that the token's credentials verifies and that it has not already been spent, the contract sends $v$ to the Ethereum destination address $addr$ (5).

### 6.5.2 Privacy-Preserving Petition

We consider the scenario where several authorities managing the country $C$ wish to issue some long-term credentials to its citizens to enable any third party to organize a privacy-preserving petition. All citizens of $C$ are allowed to participate, but should remain anonymous and unlinkable across petitions. This application extends the work of Diaz *et al.* [196] which does not consider threshold issuance of credentials.

**Chainspace petition system**. Our petition system is based on the Coconut library contract for Chainspace and a simple smart contract called 'petition'. There are three types of parties: a set of signing authorities representing $C$, a petition initiator, and

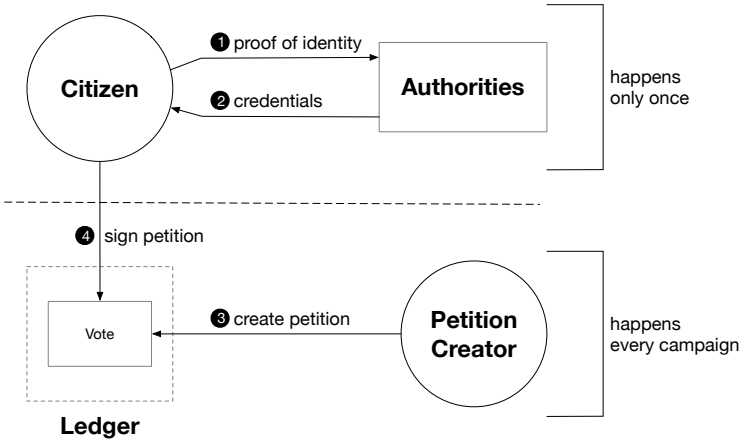


**Figure 6.6:** The petition application.

the citizens of $C$. The signing authorities create an instance of the Coconut smart contract as described in Section 6.4.1. As shown in Figure 6.6, the citizen provides a *proof of identity* to the authorities (❶). The authorities check the citizen's identity, and issue a blind and long-term signature on her private key $k$. This signature, which the citizen needs to obtain only once, acts as her long term *credential* to sign any petition (❷). Any third party can *create a petition* by creating a new instance of the petition contract and become the 'owner' of the petition. The petition instance specifies an identifier $g_s \in \mathbb{G}_1$ unique to the petition where its representation is unlinkable to the other points of the scheme[12], as well as the verification key of the authorities issuing the credentials and any application specific parameters (*e.g.* the options and current votes) (❸). In order to *sign* a petition, the citizens compute a value $\zeta = g_s^k$. They then adapt the zero-knowledge proof of the ProveCred algorithm of Section 6.2.4 to show that $\zeta$ is built from the same attribute $k$ in the credential; the petition contract checks the proofs and the credentials, and checks that the signature is fresh by verifying that $\zeta$ is not part of a spent list. If all the checks pass, it adds the citizens' signatures to a list of records and adds $\zeta$ to the spent list to prevents a citizen from signing the same petition multiple times (prevent double spending) (❹). The zero-knowledge proof ensures that $\zeta$ is built from a signed private key $k$, meaning that the users correctly executed the callback to prove that they are citizens of $C$.

[12]This identifier can be generated through a hash function $\mathbb{F}_p \rightarrow \mathbb{G}_1 : \widetilde{H}(s) = g_s \mid s \in \mathbb{F}_p$.

**Security consideration**. Coconut's blindness property prevents the authorities from learning the citizen's secret key, and misusing it to sign petitions on behalf of the citizen. Another benefit is that it lets citizens sign petitions anonymously; citizens only have to go through the issuance phase once, and can then re-use credentials multiple times while staying anonymous and unlinkable across petitions. Coconut allows for distributed credentials issuance, removing a central authority and preventing a single entity from creating arbitrary credentials to sign petitions multiple times.

### 6.5.3 Censorship-Resistant Distribution of Proxies

Proxies can be used to bypass censorship, but often become the target of censorship themselves. We present a system based on Coconut for censorship-resistant distribution of proxies (CRS). In our CRS, the volunteer $V$ runs proxies, and is known to the Coconut authorities through its long-term public key. The authorities establish reputability of volunteers (identified by their public keys) through an out of band mechanism. The user $U$ wants to find proxy IP addresses belonging to reputable volunteers, but volunteers want to hide their identity. As shown in Figure 6.7, $V$ gets an ephemeral public key $pk'$ from the proxy (❶), provides *proof of identity* to the authorities (❷), and gets a *credential* on two private attributes: the proxy IP address, $pk'$, and the time period $\delta$ for which it is valid (❸). $V$ shares the credential with the concerned proxy (❹), which creates the *proxy info* including $pk'$, $\delta$, and the credential; the proxy 'registers' itself by appending this information to the blockchain along with a zero-knowledge proof and the material necessary to verify the validity of the credential (❺). The users $U$ monitor the blockchain for proxy registrations. When a registration is found, $U$ indicates the intent to use a proxy by publishing to the blockchain a *request info* message which looks as follows: user IP address encrypted under $pk'$ which is embedded in the registration blockchain entry (❻). The proxy continuously monitors the blockchain, and upon finding a user request addressed to itself, *connects* to $U$ and presents proof of knowledge of the private key associated with $pk'$ (❼). $U$ verifies the proof, the proxy IP address and its validity period, and then starts relaying its traffic through the proxy.

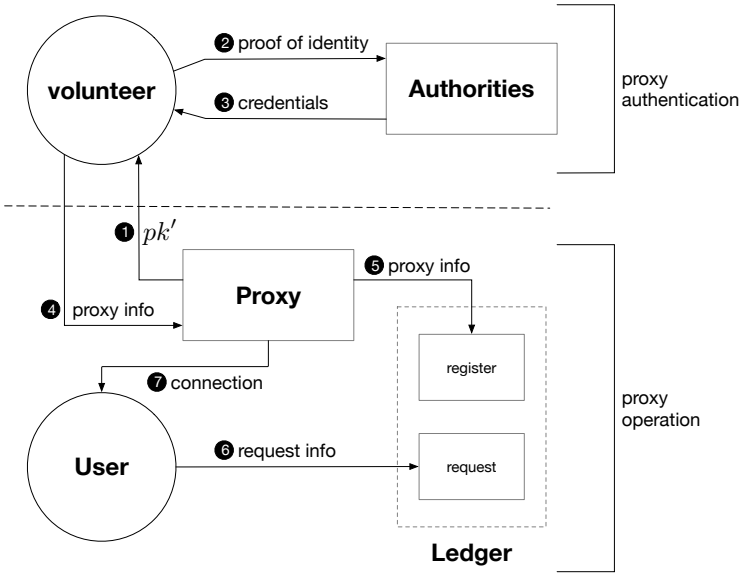


**Figure 6.7:** The censorship-resistant proxy distribution system.

**Security consideration**. A number of CRSes have been previously proposed that employ techniques such as mimicing popular unblocked protocols, tunnelling traffic through 'unblockable' protocols, and covert proxies and channels (*e.g.* user-generated content on social media platforms) [197]. A common limitation of censorship resistance schemes is relying on volunteers that are *assumed* to be resistant to coercion: either (i) the volunteer is a large, commercial organisation (*e.g.* Amazon or Google) over which the censor cannot exert its influence; and/or (ii) the volunteer is located outside the country of censorship. However, both these assumptions were proven wrong [198, 199]. The proposed CRS overcomes this limitation by offering coercion-resistance to volunteers from censor-controlled users and authorities. Due to Coconut's blindness property, a volunteer can get a credential on its IP address and ephemeral public key without revealing those to the authorities. The users get proxy IP addresses run by the volunteer, while being unable to link it to the volunteer's long-term public key. It might be hard for a censor to take down large, commercial parties—but these can be potentially forced to stop supporting the CRS [198]. Similarly, the emergence of global surveillance coalitions invalidates prevailing CRS assumptions based on the censor's geographic reach [199]. The proposed CRS relies on multiple organisations validated by a distributed set of

| Operation | Mean (ms) | Std. (ms) |
|---|---|---|
| PrepareBlindSign | 2.633 | ± 0.003 |
| BlindSign | 3.356 | ± 0.002 |
| Unblind | 0.445 | ± 0.002 |
| AggCred | 0.454 | ± 0.000 |
| ProveCred | 1.544 | ± 0.001 |
| VerifyCred | 10.497 | ± 0.002 |

**Table 6.1:** Execution times for the cryptographic primitives described in Section 6.2, measured for one private attribute over 10,000 runs. AggCred is computed assuming two authorities; the other primitives are independent of the number of authorities.

authorities, that can run proxies with complete deniability. Moreover, the authorities operate independently and can be controlled by different entities, and are resilient against a threshold number of authorities being dishonest or taken down.

## 6.6 Evaluation

We present the evaluation of the Coconut threshold credentials scheme; first we present a benchmark of the cryptographic primitives described in Section 6.2 and then we evaluate the smart contracts described in Section 6.5.

### 6.6.1 Cryptographic Primitives

We implement the primitives described in Section 6.2 in Python using petlib [136] and bplib [200]. The bilinear pairing is defined over the Barreto-Naehrig [201] curve, using OpenSSL as arithmetic backend.

**Timing benchmark**. Table 6.1 shows the mean and standard deviation of the execution of each procedure described in section Section 6.2. Each entry is the result of 10,000 runs measured on an desktop computer, 3.6GHz Intel Xeon. Signing is much faster than verifying credentials—due to the pairing operation in the latter; verification takes about 10ms; signing a private attribute is about 3 times faster.

**Communication complexity and packets size**. Table 6.2 shows the communication complexity and the size of each exchange involved in the Coconut credentials scheme, as presented in Figure 6.3. The communication complexity is expressed as a function of the number of signing authorities ($n$), and the size of each attribute is limited to

| Number of authorities: $n$, Signature size: 132 bytes | | |
|---|---|---|
| **Transaction** | **Complexity** | **Size [B]** |
| Signature on one public attribute: | | |
| ❶ request credential | $O(n)$ | 32 |
| ❷ issue credential | $O(n)$ | 132 |
| ❸ verify credential | $O(1)$ | 162 |
| Signature on one private attribute: | | |
| ❶ request credential | $O(n)$ | 516 |
| ❷ issue credential | $O(n)$ | 132 |
| ❸ verify credential | $O(1)$ | 355 |

**Table 6.2:** Communication complexity and transaction size for the Coconut credentials scheme when signing one public and one private attribute (see Figure 6.3 of Section 6.2).

32 bytes as the output of the SHA-2 hash function. The size of a credential is 132 bytes. The highest transaction sizes are to request and verify credentials embedding a private attribute; this is due to the proofs $\pi_s$ and $\pi_v$ (see Section 6.2). The proof $\pi_s$ is approximately 318 bytes and $\pi_v$ is 157 bytes.

**Client-perceived latency**. We evaluate the client-perceived latency for the Coconut threshold credentials scheme for authorities deployed on Amazon AWS [202] when issuing partial credentials on one public and one private attribute. The client requests a partial credential from 10 authorities, and latency is defined as the time it waits to receive $t$-out-of-10 partial signatures. Figure 6.8 presents measured latency for a threshold parameter t ranging from 1–10. The dots correspond to the average latency and the error-bars represent the normalized standard deviation, computed over 100 runs. The client is located in London while the 10 authorities are geographically distributed across the world; US East (Ohio), US West (N. California), Asia Pacific (Mumbai), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Tokyo), Canada (Central), EU (Frankfürt), EU (London), and South America (São Paulo). All machines are running a fresh 64-bit Ubuntu distribution, the client runs on a *large* AWS instance and the authorities run on *nano* instances. As expected, we observe that the further the authorities are from the client, the higher the latency due to higher response times; the first authorities to respond are always those situated in Europe, while Sidney and Tokyo are the latest. Latency grows linearly, with the exception of

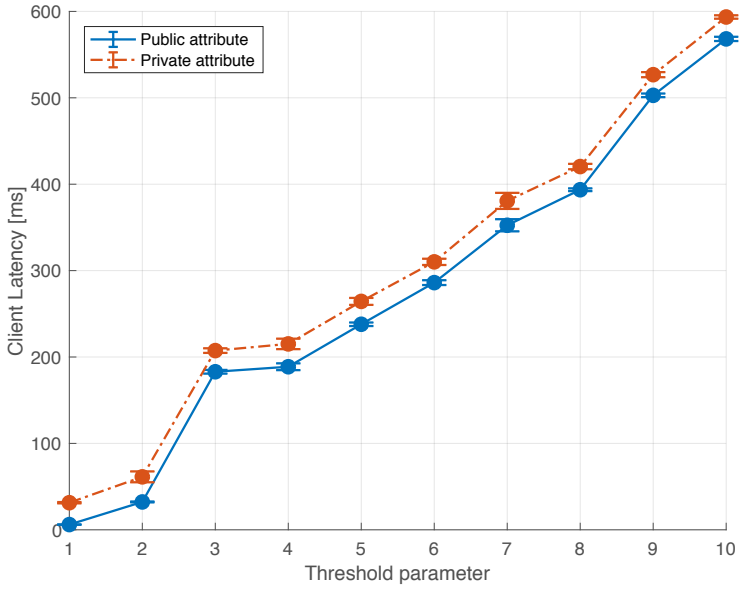


**Figure 6.8:** Client-perceived latency for Coconut threshold credentials scheme with geographically distributed authorities, measured for one attribute over 100 runs.

| Coconut smart contract library | | | |
|---|---|---|---|
| **Operation** | **Mean (ms)** | **Std. (ms)** | **Size (kB)** |
| Create [g] | 0.195 | ± 0.065 | ∼ 1.38 |
| Create [c] | 12.099 | ± 0.471 | - |
| Request [g] | 7.094 | ± 0.641 | ∼ 3.77 |
| Request [c] | 6.605 | ± 0.559 | - |
| Issue [g] | 4.382 | ± 0.654 | ∼ 3.08 |
| Issue [c] | 0.024 | ± 0.001 | - |
| Verify [g] | 5.545 | ± 0.859 | ∼ 1.76 |
| Verify [c] | 10.814 | ± 1.160 | - |

**Table 6.3:** Timing and transaction size of the Chainspace implementation of the Coconut smart contract library described in Section 6.4.1, measured for two authorities and one private attributes over 10,000 runs. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

a large jump (of about 150 ms) when $t$ increases from 2 to 3—this is due to the 7 remaining authorities being located outside Europe. The latency overhead between credential requests on public and private attributes remains constant.

### 6.6.2 Chainspace Implementation

We evaluate the Coconut smart contract library implemented in Chainspace, as well as the the coin tumbler (Section 6.5.1) and the privacy-preserving e-petition (Section 6.5.2) applications that use this library. As expected, Table 6.3 shows that

| Coin tumbler<br>**Operation** | **Mean (ms)** | **Std. (ms)** | **Size (kB)** |
|---|---|---|---|
| InitTumbler [g] | 0.235 | ± 0.065 | ∼ 1.38 |
| InitTumbler [c] | 19.359 | ± 0.773 | - |
| Pay [g] | 11.939 | ± 0.792 | ∼ 4.28 |
| Pay [c] | 6.625 | ± 0.559 | - |
| Redeem [g] | 0.132 | ± 0.012 | ∼ 3.08 |
| Redeem [c] | 11.742 | ± 0.757 | - |

**Table 6.4:** Timing and transaction size of the Chainspace implementation of the coin tumbler smart contract (described in Section 6.5), measured over 10,000 runs. The transactions are independent of the number of authorities. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

| Privacy-preserving e-petition<br>**Operation** | **Mean (ms)** | **Std. (ms)** | **Size (kB)** |
|---|---|---|---|
| InitPetition [g] | 3.260 | ± 0.209 | ∼ 1.50 |
| InitPetition [c] | 3.677 | ± 0.126 | - |
| SignPetition [g] | 7.999 | ± 0.467 | ∼ 3.16 |
| SignPetition [c] | 15.801 | ± 0.537 | - |

**Table 6.5:** Timing and transaction size of the Chainspace implementation of the privacy-preserving e-petition smart contract (described in Section 6.5), measured over 10,000 runs. The transactions are independent of the number of authorities. The notation [g] denotes the execution the procedure and [c] denotes the execution of the checker.

the most time consuming procedures are the checker of Create and the checker of Verify; i.e., they call the VerifyCred primitives which takes about 10 ms (see Table 6.1). Table 6.3 is computed assuming two authorities; the transaction size of Issue increases by about 132 bytes (*i.e.* the size of the credentials) for each extra authority[13] while the other transactions are independent of the number of authorities. Similarly, the most time consuming procedure of the coin tumbler (Table 6.4) application and of the privacy-preserving e-petition (Table 6.5) are the checker of InitTumbler and the checker of SignPetition, respectively; these two checkers call the BlindVerify primitive involving pairing checks. The Pay procedure of the coin tumbler presents the highest transaction size as it is composed of two distinct transactions: a coin transfer transaction and a Request transaction from the

[13]The Request and Issue procedures are only needed in the case of on-chain issuance (see Section 6.4.1).

| Coconut Ethereum smart contract library | | | |
|---|---|---|---|
| **Operation** | **Mean (ms)** | **Std. (ms)** | **Gas** |
| Create | 27.45 | $\pm$ 3.054 | $\sim 23,000$ |
| Verify | 120.17 | $\pm$ 25.133 | $\sim 2,150,000$ |

**Table 6.6:** Timing and gas cost of the Ethereum implementation of the Coconut smart contract library described in Section 6.4.2. Measured over 100 runs, for one public attribute. The transactions are independent of the number of authorities.

Coconut contract library. However, they are all practical, and they all run in a few milliseconds. These transactions are independent of the number of authorities as issuance is either handled off-chain or by the Coconut smart contract library.

### 6.6.3 Ethereum Implementation

We evaluate the Coconut Ethereum smart contract library described in Section 6.4.2 using the Go implementation of Ethereum on an Intel Core i5 laptop with 12GB of RAM running Ubuntu 17.10. Table 6.6 shows the execution times and gas costs for different procedures in the smart contract. The execution times for Create and Verify are higher than the execution times for the Chainspace version (Table 6.3) of the library, due to the different implementations. The arithmetic underlying Coconut in Chainspace is performed through Python naively binding to C libraries, while in Ethereum arithmetic is defined in solidity and executed by the EVM.

We also observe that the Verify function has a significantly higher gas cost than Create. This is mostly due to the implementation of elliptic curve multiplication as a native Ethereum smart contract—the elliptic curve multiplication alone costs around $1,700,000$ gas, accounting for the vast majority of the gas cost, whereas the pairing operation using the pre-compiled contract costs only 260,000 gas. The actual fiat USD costs corresponding to those gas costs, fluctuate wildly depending on the price of Ether—Ethereum's internal value token—the load on the network, and how long the user wants to wait for the transaction to be mined into a block. As of February 7th 2018, for a transaction to be confirmed within 6 minutes, the transaction fee for Verify is \$1.74, whereas within 45 seconds, the transaction fee is \$43.5.[14] The bottleneck of our Ethereum implementation is the high-level arithmetic in $\mathbb{G}_2$.

[14] `https://ethgasstation.info/`

| Scheme | Blindness | Unlinkable | Aggregable | Threshold | Size |
|---|---|---|---|---|---|
| **[114]** Waters Signature | ✗ | ✗ | ○ | ✗ | 2 Elements |
| **[115]** LOSSW Signature | ✗ | ✗ | ◒ | ✗ | 2 Elements |
| **[116]** BGLS Signature | ✓ | ✗ | ● | ✓ | 1 Element |
| **[107]** CL Signature | ✓ | ✓ | ◒ | ✗ | $O(q)$ Elements |
| **[119]** Idemix | ✓ | ✓ | ○ | ✗ | $O(q)$ Elements |
| **[123]** U-Prove | ✓ | ✓ | ○ | ✗ | $O(v)$ Elements |
| **[124]** ACL | ✓ | ✓ | ○ | ✗ | $O(v)$ Elements |
| **[109]** Pointcheval and Sanders | ✓ | ✓ | ◒ | ✗ | 2 Elements |
| **[125]** Garman *et al.* | ✓ | ✓ | - | ✗ | 2 Elements |
| **[Section 6.2]** Coconut | ✓ | ✓ | ● | ✓ | 2 Elements |

**Table 6.7:** Comparison of Coconut with other relevant cryptographic constructions. The aggregability of the signature scheme reads as follows; ○ : not aggregable, ◒ : sequentially aggregable, ● : aggregable. The signature size is measured asymptotically or in terms of the number of group elements it is made of (for constant-size credentials); $q$ indicates the number of attributes embedded in the credentials and $v$ the number of times the credential may be shown unlinkably.

However, Ethereum provides a pre-compiled contract for arithmetic operations in $\mathbb{G}_1$. We could re-write our cryptographic primitives by swapping all the operations in $\mathbb{G}_1$ and $\mathbb{G}_2$, at the cost of relying on the SXDH assumption [203] (which is stronger than the standard XDH assumption that we are currently using).

## 6.7 Comparison with Related Works

We compare the Coconut cryptographic constructions and system with related work in Table 6.7, along the dimensions of key properties offered by Coconut—blindness, unlinkability, aggregability (i.e., whether multiple authorities are involved in issuing the credential), threshold aggregation (i.e., whether a credential can be aggregated using signatures issued by a subset of authorities), and signature size (see Sections 6.1 and 6.2). Section 2.5.2 provides a detailed description of all the schemes mentioned in Table 6.7.

## 6.8 Limitations

Coconut has a number of limitations that are beyond the scope of this work, and deferred to future work. Adding and removing authorities implies to re-run the key generation algorithm—this limitation is inherited from the underlying Shamir's secret sharing protocol [178] and can be mitigated using techniques coming from

proactive secret sharing introduced by Herzberg *et al.* [179]. However, credentials issued by authorities with different key sets are distinguishable, and therefore frequent key rotation reduces the privacy provided. As any threshold system, Coconut is vulnerable if more than the threshold number of authorities are malicious; colluding authorities could create coins to steal all the coins in the buffer of the coin tumbler application (Section 6.5.1), create fake identities or censor legitimate users of the electronic petition application (Section 6.5.2), and defeat the censorship resistance of our proxy distribution application (Section 6.5.3). Note that users' privacy is still guaranteed under colluding authorities, or an eventual compromise of their keys. Implementing the Coconut smart contract library on Ethereum is expensive (Table 6.6) as Ethereum does not provide pre-compiled contracts for elliptic curve arithmetic in $\mathbb{G}_2$; re-writing our cryptographic primitives by swapping all the operations in $\mathbb{G}_1$ and $\mathbb{G}_2$ would dramatically reduce the gas cost (and be secure under SXDH [203]).

## 6.9 Chapter Summary

Existing selective credential disclosure schemes do not provide the full set of desired properties, particularly when it comes to efficiency and issuing general purpose selective disclosure credentials without sacrificing desirable distributed trust assumptions. This limits their applicability in distributed settings such as distributed ledgers.In this paper, we present Coconut—a novel scheme that supports distributed threshold issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. We provide an overview of the Coconut system, and the cryptographic primitives underlying Coconut; an implementation and evaluation of Coconut as a smart contract library in Chainspace and Ethereum, a sharded and a permissionless blockchain respectively; and three diverse and important application to anonymous payments, petitions and censorship resistance. Coconut fills an important gap in the literature and enables general purpose selective disclosure credentials—an important privacy enhancing technology—to be efficiently used in settings with no natural single trusted third party to issue them, and to interoperate with modern transparent computation platforms.

# Chapter 7

# Conclusion

This thesis proposed technologies to overcome the following limitations of blockchain technologies and smart contract platforms: (i) poor scalability, (ii) high latency, and (iii) difficulty to operate on secret values (privacy).

Chapter 3 presented Chainspace—an open, distributed ledger platform for high-integrity and transparent processing of transactions. Chainspace offers extensibility though privacy-friendly smart contracts. We presented an instantiation of Chainspace by parameterizing it with a number of 'system' and 'application' contracts, along with their evaluation. However, unlike existing smart-contract based systems such as Ethereum, it offers high scalability through sharding across nodes, while offering high auditability. As such it offers a competitive alternative to both centralized and permissioned systems, as well as fully peer-to-peer, but unscalable systems.

Chapter 4 presented the first replay attacks against cross-shard consensus protocols in sharded distributed ledgers. These attacks affect both shard-driven and client-driven consensus protocols, and allow attackers to double-spend with minimal efforts. The attacker can act independently without colluding with any nodes, and succeed even if all nodes are honest; most of the attacks work without making any assumptions on the underlying network. While addressing these attacks seems like an implementation detail, their many variants illustrate that a fundamental re-think of cross-shard commit protocols is required to protect against them. We developed Byzcuit, a new cross-shard consensus protocol merging features from shard-led and client-led consensus protocols, and withstanding replay attacks. Byzcuit can

be seen as unifying Atomix and S-BAC, into a protocol that is efficient and secure. We implemented and evaluated it on a real cloud-based testbed, showing that it can process over 1,550 tps for 10 shards while keeping latency under 1 second (using BFT-SMART as intra-shard consensus implementation), and that its capacity scales linearly with the number of shards.

Chapter 5 presented FastPay—a settlement layer based on consistent broadcast channels, rather than full consensus. The FastPay design leverages the nature of payments to allow for asynchronous payments into accounts, and optional interactions with an external FastPay to build a practical system, while providing proofs of both safely and liveness; it also proposes and evaluates a design for sharded implementation of authorities to horizontally scale and match any throughput need. The performance and robustness of FastPay is beyond and above the state of the art, and validates that moving away from both centralized solutions and full consensus to manage pre-funded retail payments has significant advantages. Authorities can jointly process tens of thousands of transactions per second (we observed a peak of 160,000 tx/sec) using merely commodity hardware and lean software. A payment confirmation latency of less than 200ms across continents make FastPay practical for point of sale payments—where goods and services need to be delivered fast and in person. Pretty much instant settlement enables retail payments to be freed from intermediaries, such as banks payment networks, since they eliminate any credit risk inherent in deferred netted end-of-day payments, that underpin today most national Fast Payment systems [173]. Further, FastPay can tolerate up to one-third of authorities crashing or even becoming Byzantine without losing either safety or liveness (or performance). This is in sharp contrast with existing centralized settlement layers operating on specialized mainframes with a primary / backup crash fail strategy (and no documented technical strategy to handle Byzantine operators). Surprisingly, it is also in contrast with permissioned blockchains, which have not achieved similar levels of performance and robustness yet, due to the complexity of engineering and scaling full byzantine fault-tolerant consensus protocols.

Chapter 6 presented Coconut—a novel scheme that supports distributed thresh-

old issuance, public and private attributes, re-randomization, and multiple unlinkable selective attribute revelations. Previous selective credential disclosure schemes do not provide the full set of desired properties, particularly when it comes to efficiency and issuing general purpose selective disclosure credentials without sacrificing desirable distributed trust assumptions. This limits their applicability in distributed settings such as distributed ledgers, and prevents security engineers from implementing separation of duty policies that are effective in preserving integrity. We provide an overview of the Coconut system, and the cryptographic primitives underlying Coconut; an implementation and evaluation of Coconut as a smart contract library in Chainspace and Ethereum, a sharded and a permissionless blockchain respectively; and three diverse and important application to anonymous payments, petitions and censorship resistance. Coconut fills an important gap in the literature and enables general purpose selective disclosure credentials—an important privacy enhancing technology—to be efficiently used in settings with no natural single trusted third party to issue them, and to interoperate with modern transparent computation platforms.

## 7.1 Future Directions

A number of future directions are left open to explore.

**Permissionless sharded systems**. Sharded distributed systems such as Chainspace raise the concern of how to map nodes to shards. In permissioned systems, this process is usually done according to the policy of a designated party, but it remains an open challenge for open systems. Similarly, the reconfiguration of sharded ledgers is also an open question; if nodes are dynamically reconfigured across shards, there needs to be a mechanism to transfer the blockchains state from one node to another without losing liveness during this process. The design of a mechanism to avoid the creation of malicious shards is also an open question. Chainspace allows contract creators to designate which shards are responsible for handling the state associated with their smart contract. It is however unclear how the system could recover if smart contract creators select malicious shards.

**Scaling intra-shard consensus**. Chainspace scales by adding new shards to the

system and effectively running multiple instances of intra-shard consensus in a coordinated fashion. However, it uses intra-shard consensus as a black box; the prototype of Byzcuit presented in Section 4.7 is implemented using BFT-SMART but any BFT consensus protocol would work as a drop-in replacement, and it works in a setting where each node is run by a distinct authority. Another research direction is to explore how a single authority could scale out by implementing its node across multiple machines (possibly a whole data center). Such a system is not trivial to design as it requires an intra-shard consensus protocol that can efficiently share its load and take advantage of multiple machines.

**Scaling execution**. Scaling smart contract execution is another open research direction. Smart contracts may contain heavy operations and execution could quickly become the bottleneck of the system, eventually slowing down consensus and harming scalability. Despite Chainspace does not execute smart contracts, it still needs to verify the correctness of transactions which could sometimes be even more expensive than re-executing the transaction (depending on the contract). A possible direction to investigate to solve this problem is the design of a general purpose zk-SNARK verifier that could be used to verify any transaction; zk-SNARKs are particularly suited for blockchains as they are typically cheap to verify (and heavy to compute). An another direction could be to scale execution across multiple machines; this could be simpler for smart contract with special semantics (*e.g.* payments, as shown by FastPay in Chapter 5) but is a challenge for general purpose smart contracts. The traditional database literature explores this path but blockchains have the option to rely on totally ordered sequence of transactions before attempting execution.

**Ensuring resilience under attack**. The key advantage of blockchain technologies is their resilience to Byzantine behavior. Blockchains operate under the assumption that a subset of the nodes can behave arbitrarily maliciously, but how to test blockchains under attack is an open question. The core of the challenge comes from allowing Byzantine nodes to perform any kind of actions (in contrast with crash-tolerant protocols where the adversary is constrained to a specific set of actions). Further, it is not enough to ensure that safety and liveness are preserved under attack, we must

also ensure that the adversary cannot significantly slow down the system and make it unpractical. This is particularly true for leader-based protocols that need to handle the possibility of malicious leaders.

**Incentive mechanism**. Another direction for future works is the integration of incentive mechanisms into open systems; sharded distributed systems raise the additional concern that any transaction fee may be diluted as it is split amongst the nodes of multiple shards, and thus may not be sufficient to incentivize honest behavior. Closely related to that issue, preventing denial of service attacks against sharded ledgers is a challenge; sharded consensus usually involve a high number of communications (coming from both the intra-shard and inter-shard consensus protocols) which are potential vectors for denial-of-service attacks.

**Reconfiguration without consensus**. Efficiently replacing a set of authorities in the absence of consensus remains an open challenge; this applies to both FastPay and Coconut, as well as to any distributed system that does not implement consensus.

## 7.2 Closing Thoughs

We showed that it is now possible to build secure and scalable distributed ledgers that can accommodate high throughput. We demonstrated that blockchain technologies can be brought to retail payment system through the use of extremely low-latency distributed side-infrastructures. Further, we showed how to design distributed ledgers with native support for privacy-preserving applications, and how to use them to issue credentials in a blockchain setting; this enables a number of novel distributed applications. The technologies described in this thesis can be implemented to increase the fairness, robustness, efficiency, and privacy of current payment systems, while decreasing their costs.

# Bibliography


[1] Denis Roio, Francesca Bria, James Barritt, Jaap-Henk Hoepman, Mark de Villiers, Priya Samuel, George Danezis, Tom Demeyer, Shehar Bano, and Oleguer Sagarra. DECODE Whitepaper v1.0. https://decodeproject.github.io/whitepaper, 2018.

[2] Interchain Foundation. ICF Q22020 Funding Recipients. https://interchain-io.medium.com/icf-q2-2020-funding-recipients-e5cbb326c23c, 2020.

[3] Nym. Nym. https://nymtech.net, 2020.

[4] Eleftherios Kokoris Kogias, Philipp Svetolik Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Alexander Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE Symposium on Security and Privacy*, 2018.

[5] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling Blockchain via Full Sharding. In *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[6] Harmony Team. Harmony Technical Whitepaper Version 2.0. https://harmony.one/whitepaper.pdf, 2019.

[7] CoinMarketCap. Global Charts: Total Market Capitalization. https://coinmarketcap.com/charts, 2020.

[8] Bank of England. Digital Currencies. http://www.bankofengland.co.uk/research/Pages/onebank/cbdc.aspx, 2018.

[9] CoinDesk. Bank of America, Microsoft Partner on Blockchain Trade Finance. https://www.coindesk.com/bank-america-microsoft-partner-blockchain-trade-finance, 2016.

[10] International Monetary Fund. Fintech and Financial Services : Initial Considerations. http://www.imf.org/en/Publications/Staff-Discussion-Notes/Issues/2017/06/16/Fintech-and-Financial-Services-Initial-Considerations-44985, 2017.

[11] followmyvote.com. FollowMyVote. https://followmyvote.com, 2020.

[12] ProtocolLab. A Robust Foundation for Humanitys Information. https://filecoin.io, 2020.

[13] Michał Król, Alberto Sonnino, Argyrios G. Tasiopoulos, Ioannis Psaras, and Etienne Rivière. PASTRAMI: Privacy-preserving, Auditable, Scalable & Trustworthy Auctions for Multiple Items. In *ACM/IFIP Middleware*, 2020.

[14] Alberto Sonnino, Michał Król, Argyrios G Tasiopoulos, and Ioannis Psaras. AStERISK: Auction-based Shared Economy ResolutIon System for blocKchain. In *Workshop on Decentralized IoT Systems and Security*, 2019.

[15] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[16] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998.

[17] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[18] Digital Trends. The World's Cryptocurrency Mining Uses more Electricity than Iceland. https://www.digitaltrends.com/computing/

bitcoin-ethereum-mining-use-significant-electrical-power, 2017.

[19] Visa. Visa Acceptance for Retailers. https://usa.visa.com/run-your-business/small-business-tools/retail.html, 2020.

[20] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *USENIX Conference on Networked Systems Design and Implementation*, 2016.

[21] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On Scaling Decentralized Blockchains. In *Financial Cryptography Workshop on Bitcoin and Blockchain Research*, 2016.

[22] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the Age of Blockchains. In *ACM Conference on Advances in Financial Technologies*, 2019.

[23] Ben Laurie. Certificate Transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[24] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger EIP-150 Revision. http://gavwood.com/paper.pdf, 2016.

[25] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. In *Network and Distributed System Security Symposium*, 2018.

[26] Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. In *Network and Distributed Systems Security Symposium*, 2019.

[27] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay Attacks and Defenses Against Cross-shard Consensus in Sharded Distributed Ledgers. In *IEEE European Symposium on Security and Privacy*, 2020.

[28] Mathieu Baudet, George Danezis, and Alberto Sonnino. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *ACM Conference on Advances in Financial Technologies*, 2020.

[29] Mustafa Al-Bassam, Alberto Sonnino, Michał Król, and Ioannis Psaras. Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations. arXiv preprint arXiv:1805.06411, 2018.

[30] Michał Król, Alberto Sonnino, Mustafa Al-Bassam, Argyrios Tasiopoulos, and Ioannis Psaras. Proof-of-Prestige: A Useful Work Reward System for Unverifiable Tasks. In *IEEE International Conference on Blockchain and Cryptocurrency*, 2019.

[31] George Danezis and Alberto Sonnino. SybilQuorum: Open Distributed Ledgers Through Trust Networks. arXiv preprint arXiv:1906.12237, 2019.

[32] Alberto Sonnino. FMPC: Secure Multiparty Computation from Fourier Series and Parseval's Identity. arXiv preprint arXiv:1912.02583, 2019.

[33] Christos Andrikos, Lejla Batina, Lukasz Chmielewski, Liran Lerman, Vasilios Mavroudis, Kostas Papagiannopoulos, Guilherme Perin, Giorgos Rassias, and Alberto Sonnino. Location, location, location: Revisiting modeling and exploitation for location-based side channel leakages. In *Asiacrypt*, 2019.

[34] Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. EL PASSO: Privacy-preserving, Asynchronous Single Sign-On. International Symposium on Privacy Enhancing Technologies, 2021.

[35] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: White-Glove Approach for BFT Testing. arXiv preprint arXiv:2004.10617, 2020.

[36] Mustafa Al-Bassam, Alberto Sonnino, Vitalik Buterin, and Ismail Khoffi. Fraud Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities. In *Financial Cryptography and Data Security*, 2021.

[37] Juan A. Garay and Aggelos Kiayias. SoK: A Consensus Taxonomy in the Blockchain Era. IACR Cryptology ePrint Archive, 2018.

[38] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[39] Christian Cachin. Architecture of the Hyperledger Blockchain Fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, 2016.

[40] Calibra. The Libra Blockchain. `https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf`, 2019.

[41] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State Machine Replication in the Libra Blockchain, 2019.

[42] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2013.

[43] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *IEEE Security and Privacy)*, 2014.

[44] Christian Cachin and Marko Vukolić. Blockchain Consensus Protocols in the Wild. arXiv preprint arXiv:1707.0187, 2017.

[45] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[46] Team Rocket. Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. `https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV`, 2018.

[47] John R. Douceur. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems*, 2002.

[48] Vitalik Buterin. The Meaning of Decentralization. `https://medium.com/@VitalikButerin/the-meaning-of-decentralization-a0c92b76a274`, 2017.

[49] Carmela Troncoso, Marios Isaakidis, George Danezis, and Harry Halpin. Systematizing Decentralization and Privacy: Lessons from 15 Years of Research and Deployments. In *International Symposium on Privacy Enhancing Technologies*, 2017.

[50] Miguel Castro and Barbara Liskov. Byzantine Fault Tolerance. `https://patents.google.com/patent/US6671821B1`, 2003.

[51] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communicationsof the ACM*, 21(7):558–565, 1978.

[52] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, 1978.

[53] Dale Skeen. Nonblocking Commit Protocols. In *ACM SIGMOD International Conference on Management of Data*, 1981.

[54] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Information and Computation*, 1995.

[55] Fred B Schneider. Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial. In *ACM Computing Surveys*, 1990.

[56] 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, 1985.

[57] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[58] Brian M Oki and Barbara H Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-available Distributed Systems. In *ACM Symposium on Principles of Distributed Computing*, 1988.

[59] Diego Ongaro and John K Ousterhout. In search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*, 2014.

[60] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance Broadcast for Primary-backup Systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011.

[61] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing*, 2019.

[62] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. `https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf`, 2016.

[63] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. arXiv preprint arXiv:1710.09437, 2017.

[64] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[65] Cardano. Cardano. https://www.cardano.org, 2020.

[66] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably Secure Proof-of-Stake Blockchain Protocol. In *International Cryptology Conference*, 2017.

[67] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. In *Eurocrypt*, 2018.

[68] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *International Cryptology Conference*, 1992.

[69] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Crypto*, 1992.

[70] Adam Back. A Partial Hash Collision Based Postage Scheme. http://www.hashcash.org/papers/announce.txt, 1997.

[71] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. In *Eurocrypt*, 2015.

[72] Arvind Narayanan and Jeremy Clark. Bitcoin's Academic Pedigree. *ACM Queue*, 15(4):20–49, 2017.

[73] Phil Daian, Rafael Pass, and Elaine Shi. Snow White: Provably Secure Proofs of Stake. IACR Cryptology ePrint Archive, 2016.

[74] Ethereum Blog. Introducing Casper "the Friendly Ghost". https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/, 2015.

[75] Ian Grigg. EOS Whitepaper. https://whitepaperdatabase.com/eos-whitepaper, 2017.

[76] Evangelos Deirmentzoglou, Georgios Papakyriakopoulos, and Constantinos Patsakis. A Survey on Long-Range Attacks for Proof of Stake Protocols. *IEEE Access*, 7(13):28712–28725, 2019.

[77] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-Bleeding Attacks on Proof-of-Stake Blockchains. In *Crypto Valley Conference on Blockchain Technology*, 2018.

[78] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol for Open Blockchains. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[79] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine Replication under Attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[80] Leemon Baird. Hashgraph Consensus: Fair, Fast, Byzantine Fault Tolerance. *Swirlds Tech Report, Tech. Rep.*, 2016.

[81] George Danezis and David Hrycyszyn. Blockmania: from Block DAGs to Consensus. arXiv preprint arXiv:1809.01620, 2018.

[82] Ramakrishna Kotla and Mike Dahlin. High Throughput Byzantine Fault Tolerance. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2004.

[83] Benjamin Wester, James A. Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[84] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computing Systems*, 27(4):7:1–7:39, 2009.

[85] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. In *European Conference on Computer Systems*, 2010.

[86] 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*, 2016.

[87] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. XFT: Practical Fault Tolerance beyond Crashes. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[88] Giulio Prisco. Intel Develops 'Sawtooth Lake' Distributed Ledger Technology for the Hyperledger Project. *Bitcoin Magazine*, 2016.

[89] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *ACM Symposium on Operating Systems Principles*, 2003.

[90] Marko Vukolić. Rethinking Permissioned Blockchains. In *ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, 2017.

[91] George Danezis and Sarah Meiklejohn. Centrally Banked Cryptocurrencies. In *Network and Distributed System Security Symposium*, 2016.

[92] Vitalik Buterin. Cross-shard Contract Yanking. `https://ethresear.ch/t/cross-shard-contract-yanking/1450`, 2018.

[93] Lorenzo Alvisi, Allen Clement, Alessandro Epasto, Silvio Lattanzi, and Alessandro Panconesi. SoK: The Evolution of Sybil Defense via Social Networks. In *IEEE Symposium on Security and Privacy*, 2013.

[94] David Mazieres. The Stellar Consensus Protocol: A federated Model for Internet-Level Consensus. `http://citeseerx.ist.psu.edu/viewdoc/`

download?doi=10.1.1.696.93&amp=&rep=rep1&amp=&type=pdf, 2015.

[95] 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 *USENIX Security Symposium*, 2016.

[96] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *ACM Symposium on Operating Systems Principles*, pages 51–68, 2017.

[97] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[98] Satoshi Nakamoto. Bitcoin Open Source Implementation of P2P Currency. http://p2pfoundation.ning.com/forum/topics/bitcoin-open-source, 2009.

[99] E-Gold. E-Gold. https://cs.stanford.edu/people/eroberts/cs201/projects/2010-11/Bitcoins/e-gold.html, 2020.

[100] Tor Project. Browse Privately. Explore Freely. https://www.torproject.org, 2020.

[101] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computing Systems*, 31(3):8:1–8:22, 2013.

[102] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *ACM Symposium on Operating Systems Principles*, pages 15–28, 2011.

[103] Baruch Awerbuch and Christian Scheideler. Robust Random Number Generation for Peer-to-peer Systems. In *International Conference on Principles of Distributed Systems*, 2006.

[104] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable Bias-Resistant Distributed Randomness. In *IEEE Symposium on Security and Privacy*, 2017.

[105] Long Hoang Le, Carlos Eduardo Bezerra, and Fernando Pedone. Dynamic Scalable State Machine Replication. In *ACM Symposium on Operating Systems Principles*, 2016.

[106] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An Incentive-Compatible Cryptocurrency Based on Permissionless Byzantine Consensus. `https://download.massnet.org/research/paper10`, 2016.

[107] Jan Camenisch and Anna Lysyanskaya. Signature Schemes and Anonymous Credentials from Bilinear Maps. In *International Cryptology Conference*, 2004.

[108] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. MAcs and Keyed-Verification Anonymous Credentials. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[109] David Pointcheval and Olivier Sanders. Short Randomizable Signatures. In *Cryptographers Track at the RSA Conference*, 2016.

[110] Stefan A Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. Mit Press, 2000.

[111] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *International Conference on the Theory and Application of Cryptographic Techniques*, 1986.

[112] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. Pairings for Cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[113] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. In *Asiacrypt*, 2001.

[114] Brent Waters. Efficient Identity-Based Encryption Without Random Oracles. In *Eurocrypt*, 2005.

[115] Steve Lu, Rafail Ostrovsky, Amit Sahai, Hovav Shacham, and Brent Waters. Sequential Aggregate Signatures, Multisignatures, and verifiably Encrypted Signatures without Random Oracles. *Journal of cryptology*, 26(2):340–373, 2013.

[116] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Eurocrypt*, 2003.

[117] Alexandra Boldyreva. Efficient Threshold Signature, Multisignature and Blind Signature Schemes Based on the Gap-Diffie-Hellman-group Signature Scheme. IACR Cryptology ePrint Archive, 2002.

[118] Kwangsu Lee, Dong Hoon Lee, and Moti Yung. Aggregating CL-Signatures Revisited: Extended Functionality and Better Efficiency. In *Financial Cryptography and Data Security*, 2013.

[119] Patrik Bichsel, Carl Binding, Jan Camenisch, Thomas Groß, Tom Heydt-Benjamin, Dieter Sommer, and Greg Zaverucha. Cryptographic Protocols of the Identity Mixer Library. Technical Report, IBM, 2009.

[120] Liqun Chen, Dan Page, and Nigel P Smart. On the Design and Implementation of an Efficient DAA Scheme. In *International Conference on Smart Card Research and Advanced Applications*, 2010.

[121] David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel P Smart, and Bogdan Warinschi. Anonymous Attestation with User-Controlled Linkability. *International Journal of Information Security*, 12(3):219–249, 2013.

[122] Sébastien Canard, David Pointcheval, Olivier Sanders, and Jacques Traoré. Divisible E-cash Made Practical. In *Workshop on Public Key Cryptography*, 2015.

[123] Christian Paquin and Greg Zaverucha. U-Prove Cryptographic Specification v1.1. *Technical Report, Microsoft Corporation*, 2011.

[124] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous Credentials Light. In *ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[125] Christina Garman, Matthew Green, and Ian Miers. Decentralized Anonymous Credentials. In *Network and Distributed System Security Symposium*, 2014.

[126] Google. Cloud Spanner. `https://cloud.google.com/spanner`, 2020.

[127] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, and Jens Groth. Efficient Zero-Knowledge Proof Systems. In *Foundations of Security Analysis and Design*, 2016.

[128] George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square Span Programs with Applications to Succinct NIZK Arguments. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2014.

[129] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. BigchainDB: a Scalable Blockchain Database. `https://mycourses.aalto.fi/pluginfile.php/378362/mod_resource/content/1/bigchaindb-whitepaper.pdf`, 2016.

[130] Jerome H Saltzer, David P Reed, and David D Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[131] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *International Cryptology Conference*, 1991.

[132] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[133] Luc Lauwers and Marleen Willekens. Five Hundred Years of Bookkeeping: a Portrait of Luca Pacioli. *Tijdschrift voor Economie en Management*, 39(3):289–304, 1994.

[134] Alfredo Rial and George Danezis. Privacy-Preserving Smart Metering. In *ACM workshop on Privacy in the electronic society*, 2012.

[135] Marek Jawurek, Martin Johns, and Florian Kerschbaum. Plug-In Privacy for Smart Metering Billing. In *International Symposium on Privacy Enhancing Technologies*, 2011.

[136] George Danezis. petlib. `https://github.com/gdanezis/petlib`, 2017.

[137] Vitalik Buterin, Jeff Coleman, and Matthew Wampler-Doty. Notes on Scalable Blockchain Protocols (verson 0.3.2). `https://pdfs.semanticscholar.org/ae5b/c3aaf0e02a42f4cd41916072c87db0e04ac6.pdf`, 2015.

[138] Richard Gendal Brown. Corda: an Introduction. `https://www.corda.net/content/corda-platform-whitepaper.pdf`, 2018.

[139] Rsk. RSK: Smart contracts for Bitcoin. `http://www.rsk.co`, 2017.

[140] David D Clark and David R Wilson. A Comparison of Commercial and Military Computer Security Policies. In *IEEE Symposium on Security and Privacy*, 1987.

[141] LM Goodman. Tezos A self-amending crypto-ledger White paper. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf, 2014.

[142] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. *Distributed computing*, 11(4):203–213, 1998.

[143] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[144] Bank for International Settlements. New Developments in Large-Value Payment Systems. https://www.bis.org/cpmi/publ/d67.pdf, 2005.

[145] European Central Bank. Single Shared Platform User Detailed Functional Specifications Core Services 1st Book (Version 12.01). https://www.ecb.europa.eu/paym/target/target2/profuse/nov_2018/shared/pdf/T2_UDFS_book_1_v12.01.pdf, 2018.

[146] Swift. SWIFT The Global Provider of Secure Financial Messaging Services. https://www.swift.com, 2020.

[147] Stephen Lindsay. ISO 20022 and Real-Time Domestic Payments. *Journal of Payments Strategy & Systems*, 9(1):22–29, 2015.

[148] Susan Herbst-Murphy. Clearing and Settlement of Interbank Card Transactions: A MasterCard Tutorial for Federal Reserve Payments Analysts. https://www.philadelphiafed.org/-/media/consumer-finance-institute/payment-cards-center/publications/discussion-papers/2013/D-2013-October-Clearing-Settlement.pdf, 2013.

[149] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *International Workshop on Open Problems in Network Security*, 2015.

[150] Blockchain Council. Permissioned and Permissionless Blockchains: a Comprehensive Guide. https://www.blockchain-council.org/blockchain/permissioned-and-permissionless-blockchains-a-comprehensive-guide, 2019.

[151] Elizabeth Lopatto. Libra, Explained. Move fast and bank things. https://www.theverge.com/2019/6/26/18716326/facebook-libra-cryptocurrency-blockchain-irs-starbucks, 2019.

[152] Alex Prut. Libra Quick Introduction. https://medium.com/coinmonks/libra-quick-introduction-6ce2c51d703c, 2020.

[153] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The Consensus Number of a Cryptocurrency. In *Symposium on Principles of Distributed Computing*, 2019.

[154] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. Two-Phase Commit Optimizations in a Commercial Distributed Environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995.

[155] Butler W. Lampson and David B. Lomet. A New Presumed Commit Optimization for Two Phase Commit. In *International Conference on Very Large Data Bases*, 1993.

[156] Runchao Han, Gary Shapiro, Vincent Gramoli, and Xiwei Xu. On the Performance of Distributed Ledgers for Internet of Things. *Internet of Things*, page 100087, 2019.

[157] Hyperledger. Hyperledger Fabric V0.6. https://readthedocs.org/projects/fabricdocs/downloads/pdf/origin-v0.6, 2017.

[158] Hyperledger. Hyperledger Fabric V1.0. https://readthedocs.org/projects/hyperledger-fabric/downloads/pdf/master, 2020.

[159] Ryan Chard. Ripple Documentation. https://buildmedia.readthedocs.org/media/pdf/ripple/latest/ripple.pdf, 2018.

[160] R3. R3 Corda (Release Notes). https://docs.corda.net/releases/release-V3.2/release-notes.html, 2018.

[161] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, and Ali Bou Nassif. Performance Analysis of Hyperledger Fabric Platforms. *Security and Communication Networks*, 2018, 2018.

[162] Hyojeong Lee, Jeff Seibert, Md. Endadul Hoque, Charles Edwin Killian, and Cristina Nita-Rotaru. Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations. In *IEEE International Conference on Distributed Computing Systems*, 2014.

[163] Morten L Bech and Bart Hobijn. Technology Diffusion within Central Banking: the Case of Real-Time Gross Settlement. https://www.newyorkfed.org/research/staff_reports/sr260.html, 2006.

[164] Will Dison Andrew Dent. The Bank of England's Real-Time Gross Settlement infrastructure. https://ideas.repec.org/a/boe/qbullt/0084.html, 2012.

[165] Maurice Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[166] R3. Sizing and Performance, 2018.

[167] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. Performance Evaluation of the Quorum Blockchain Platform. arXiv preprint arXiv:1809.03421, 2018.

[168] Oumnia El Khazzani. Creating the Future of Blockchain – Thorchain Update 002. https://www.swishlabs.com/blog/creating-the-future-of-blockchain-thorchain-update-002, 2019.

[169] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network. https://lightning.network/lightning-network-paper.pdf, 2016.

[170] Pavel Prihodko, Slava Zhigulin, Mykola Sahno, Aleksei Ostrovskiy, and Olaoluwa Osuntokun. Flare: An Approach to Routing in Lightning Network. https://pdfs.semanticscholar.org/4392/166a1194010c844ec915694fd5c56da94301.pdf, 2016.

[171] Cyril Grunspan and Ricardo Pérez-Marco. Ant routing algorithm for the Lightning Network. arXiv preprint arXiv:1807.00151, 2018.

[172] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. Routing Cryptocurrency with the Spider Network. arXiv preprint arXiv:1809.05088, 2018.

[173] Stephanie Bolt, David Emery, and Paul Harrigan. Fast Retail Payment Systems. https://rba.gov.au/publications/bulletin/2014/dec/pdf/bu-1214-6.pdf, 2014.

[174] Adam Back, M Corallo, L Dashjr, M Friedenbach, G Maxwell, A Miller, A Poelstra, J Timón, and P Wuille. Enabling Blockchain Innovations with Pegged Sidechains. https://blockstream.com/sidechains.pdf, 2014.

[175] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed Key Generation in the Wild. IACR Cryptology ePrint Archive, 2012. https://eprint.iacr.org/2012/377.

[176] Jan Camenisch and Markus Stadler. Proof Systems for General Statements about Discrete Logarithms. Technical Report, ETH Zürich, 1997.

[177] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. In *Eurocrypt*, 1999.

[178] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[179] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive Secret Sharing or: How to Cope with Perpetual Leakage. In *International Cryptology Conference*, 1995.

[180] Anna Lysyanskaya, Ronald L Rivest, Amit Sahai, and Stefan Wolf. Pseudonym Systems. In *International Workshop on Selected Areas in Cryptography*, 1999.

[181] Vitalik Buterin and Christian Reitwiessner. Ethereum Improvement Proposal 197 - Precompiled Contracts for Optimal Ate Pairing Check on the Elliptic Curve alt_bn128. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-197.md`, 2017.

[182] Christian Reitwiessner. Ethereum Improvement Proposal 196 - Precompiled Contracts for Addition and Scalar Multiplication on the Elliptic Curve alt_bn128. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md`, 2017.

[183] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In *Financial Cryptography and Data Security*, 2014.

[184] Luke Valenta and Brendan Rowan. Blindcoin: Blinded, Accountable Mixes for Bitcoin. In *Financial Cryptography and Data Security*, 2015.

[185] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. TumbleBit: An Untrusted Bitcoin-Compatible Anonymous Payment Hub. In *Network and Distributed System Security Symposium*, 2016.

[186] Gregory Maxwell. CoinJoin: Bitcoin Privacy for the Real World. `https://bitcointalk.org/index.php?topic=279249`, 2013.

[187] Tim Ruffing and Pedro Moreno-Sanchez and Aniket Kate. CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin. In *European Symposium on Research in Computer Security*, 2014.

[188] George Bissias, A. Pinar Ozisik, Brian N. Levine, and Marc Liberatore. Sybil-Resistant Mixing for Bitcoin. In *Workshop on Privacy in the Electronic Society*, 2014.

[189] Sarah Meiklejohn and Rebekah Mercer. Möbius: Trustless Tumbling for Transaction Privacy. In *International Symposium on Privacy Enhancing Technologies*, 2018.

[190] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An Empirical Analysis of Anonymity in Zcash. *USENIX Security Symposium*, 2018.

[191] Sean Bowe. Cultivating Sapling: Faster zk-SNARKs. `https://z.cash/blog/cultivating-sapling-faster-zksnarks`, 2017.

[192] The Zcash Foundation. Announcing the world's largest multi-party computation ceremony. `https://z.cash/technology/paramgen/`, 2017.

[193] David Chaum, Amos Fiat, and Moni Naor. Untraceable Electronic Cash. In *Conference on the Theory and Application of Cryptography*, 1988.

[194] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for Confidential Transactions and More. In *IEEE Symposium on Security and Privacy*, 2018.

[195] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-Spending Fast Payments in Bitcoin. In *ACM conference on Computer and communications security*, 2012.

[196] Claudia Diaz, Eleni Kosta, Hannelore Dekeyser, Markulf Kohlweiss, and Girma Nigusse. Privacy Preserving Electronic Petitions. *Identity in the Information Society*, 1(1):203–219, 2008.

[197] Sheharbano Khattak, Tariq Elahi, Laurent Simon, Colleen Swanson, Steven J. Murdoch, and Ian Goldberg. SoK: Making Sense of Censorship Resistance

Systems. In *International Symposium on Privacy Enhancing Technologies*, 2016.

[198] The Tor Project. Meek-Google Suspended for Terms of Service Violations (How to Set Up Your Own), 2016. https://lists.torproject.org/pipermail/tor-talk/2016-June/041699.html.

[199] The Guardian. History of 5-Eyes Explainer. http://www.theguardian.com/world/2013/dec/02/history-of-5-eyes-explainer, 2013.

[200] George Danezis. bplib. https://github.com/gdanezis/bplib, 2017.

[201] K. Kasamatsu. Barreto-Naehrig Curves. https://tools.ietf.org/id/draft-kasamatsu-bncurves-01.html, 2014.

[202] Inc. Amazon Web Services. AWS Whitepapers. https://aws.amazon.com/whitepapers, 2017.

[203] Somindu C Ramanna and Palash Sarkar. Efficient Adaptively Secure IBBE from the SXDH Assumption. *IEEE Transactions on Information Theory*, 62(10):5709–5726, 2016.